CRANFIELD UNIVERSITY

Gergely Bándi

Virtual living organism

A rapid prototyping tool to emulate biology

School of Applied Sciences

PhD
Academic Year: 2008 – 2011

Supervisor: Prof. Jeremy J. Ramsden
October, 2011

CRANFIELD UNIVERSITY

School of Applied Science

PhD THESIS

Academic Year 2008 - 2011

Gergely Bándi

Virtual living organism

A rapid prototyping tool to emulate biology

Supervisor: Prof. Jeremy J. Ramsden

October, 2011

© Cranfield University 2011. All rights reserved. No part of this publication may be reproduced without the written permission of the copyright owner.

ABSTRACT

Rapid prototyping tools exist in many fields of science and engineering, but are rare in biology especially not general tools that can handle the diversity and complexity of the many spatial and temporal scales in nature. In this thesis a general use, cell-based, middle-out biology emulation programming framework (outlining a programming paradigm) is presented, that enables biologists to emulate and use virtual biological systems of previously unimaginable complexity and potentially get results accurate enough to be used in research and ultimately, in clinical practice, such as diagnosis or operations. With this technology, virtual organisms can be created that are viable, fit and can be optimised for any task that arises. The tool, realised with a programming framework created for the C++ language is detailed and demonstrated through several examples of increasing complexity, namely several example organisms and a cancer emulation, showing both viable virtual organisms and usable experimental results.

Keywords:

Biologically inspired algorithm, cell-based, middle-out, multi-scale, in-vivo, artificial life

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor, Professor Jeremy J. Ramsden for his support in more ways than I can count. He helped by not just defining the path, but also assisted in walking it, while creating the scenery with being a partner in endless scientific discussions I'm so found of.

I would also like to thank the LINK research group at Semmelweis University in Budapest, Hungary for their help with biological facts, details and commenting on my work. I'm very grateful to Professor Ramsden's research group: Farah, Mohammad, Termeh, Julie, Alastair and Mike at Cranfield for their continuous help with ideas and corrections, and also for making my life at Cranfield happier. Special thanks to Michael Farnsworth for all his support with proofreading, technical help and advice about how to be a more successful researcher. I would also like to thank all the friends and colleagues at Cranfield for everything they've done for me on a professional and personal level.

I wouldn't have gotten this far without the support of my parents, who also were kind enough to listen to my research, even when I suspect they haven't understood much of it. I would like to thank my girlfriend Klári for her enthusiastic help and support. Lastly, I would also like to acknowledge the support of some of my best friends, Zsolt, Milán, Tamás and Péter. Sometimes a kind word is all it takes to help you keep going.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Motivation

Computer science and software engineering have caused a boost in both scientific and general productivity. The limitation of this increase was always the computational capacity of high-end computers, but even mankind's first trip to the Moon was enabled by computers not even as powerful as today's pocket calculators. One reason that, even in this limited environment, computer science helped the world is due to its massive use of abstraction. With this technique, sometimes even the most complex of tasks can be made manageable with limited resources while giving useable results. While abstraction has been used throughout most fields of science, biology so far has remained almost exclusively with true-to-nature simulations which only use minimal amounts of abstraction and are very limited in complexity. Similarly to other disciplines, biological sciences use tests and simulations to provide evidence for theories, gather explanatory results and it is always a preoccupation to provide sufficient data for statistical analysis. Biological tests can, however, be expensive, hard to measure, inaccurate and time consuming. As in almost every other field of natural science, computer simulations are also used as a viable, cheaper alternative to physical tests.

The many scales of biology, from intracellular to ecosystemic make it difficult, if not impossible, to find a simulation tool that can be used generally, and can handle events that affect many spatial and time scales. Most simulations are created as custom software solutions to a degree that limits the reusability and interoperability of many existing tools. The existing frameworks have their special field of focus where they are optimised to give the best results; real multi-scale simulation tools are very limited. Part of the possible cause of this lies in the fact that these tools are optimised to be as accurate as possible in their respective fields and if this level of accuracy is kept through all the levels of biology, the complexity of the resulting simulation would vastly exceed today's computing capabilities.

If such a general simulation tool existed in biology, it would enable more cooperation between different disciplines and would make possible, amongst other things, large scale simulations such as in vivo simulation of the effects of different diseases, therapies or medicines on entire organisms.

1.2 Background

The goal of this research is to create a fast prototyping tool (“the framework”) that can handle both the diversity of biology and the exceedingly large complexity that comes from the multiple scales that work together inside even one organism. To address the complexity issue, a step back needs to be taken from the current paradigm of building upon a simulation that uses a mathematically correct representation of biology with whatever (still small) amount of accurate information we have from experiments, and instead use emulation as a base. Emulation aims to give the same output for the same input without representing inner workings with great detail. It essentially captures the basic idea of how biology works and applies it to the computer science-generated parts. This way the mechanisms that are believed to play no real part in the studied phenomenon can be effectively turned off, thus the complexity decreases and the simulation is much faster. Similarly the object of the experiment can be as detailed as required or is beneficial, hence providing results as accurate as needed with minimal resource requirements.

From this point onwards throughout the thesis the word “simulation” will be used at some points where the meaning is closer to the goal of a computerised version of experiments, but the execution will always be done with emulation, not simulation in a “classical” sense. It is important to note that the terms simulation and emulation are often confused in the literature, and defined in the opposite way. In this thesis, I will use the definitions given above, where emulation differs from simulation with its dynamic abstractness level, where several parts can be more abstract than others depending on the requirement, like a black box approach that is applied to different levels at the same time. The choice of calling this approach emulation is also supported by the fact that

the word is used every day in the context of a person emulating another person, where the meaning is not that they try to look the same, much less try to be the same inside, but try to show similar behaviour; this matches my definition of emulation.

Instead of defining a baseline that would either be the smallest or the largest scale emulated a **middle-out** approach will be used. This way, starting with a middle level that statistically would be used in more simulations than an extreme level, a more balanced system will be created that can be extended both ways to any required level. This **middle-out** approach is also used in many existing biological simulations (Walker & Southgate, 2009). The basic level will be the cellular level, which is considered to be the basic unit of life by many (Walker & Southgate, 2009).

Aside from all the results of the emulations created with this framework, the structure and contents of the emulation itself can be useful. In computer science biologically inspired algorithms use similar techniques to grasp the abstract meaning of biological phenomena and use the results to solve real life problems, like optimisation. The better known ones are evolutionary algorithms (Goldberg, 1989), artificial neural networks (Fausett, 1994) and cellular automata (Wolfram, 1983). Emulations created with our framework can potentially be the inspiration for such algorithms that capture the knowledge resulting from millions of years of evolution in biology. Although evolution itself is considered by some to be one of the main features of biology, our framework currently does not use this mechanism. As described by Sommerhoff (1950), evolution is a kind of adaptation involving phylogenetic modifications. Since normally the time-scale for our emulations (i.e. number of generations) would be built for is less than what is required for evolution to make noticeable changes, only the changes brought on by other, faster modes of adaptation are expected to be evident. As such, the theory behind evolutionary computation will not be present in these emulations, however a similar mechanism can be incorporated that will ensure that in the future, the operation of the emulation is optimised (Chapter 6.3).

The physical result of this research is a C++ library that emulates biology, but this library cannot work on its own and does not have self-contained functionality. A user program that uses this library creates this functionality, so the library becomes a framework. Pointing beyond the actual tools that it provides, it leaves the standard thought-patterns of object-oriented programming and outlines (in this instance it even defines) a new programming paradigm.

1.3 Related work

We are aware of no existing tools to perform general multi-scale biological emulations. However there are many projects that aim to achieve comparable properties. Related works are best viewed in the order of the main common part, biological emulations from the software foundation point of view and multi-scale biological simulations from the usage point of view.

Most of what can be called, or originated from biological emulations is found in the field of artificial intelligence (AI). AI algorithms are used in a wide variety of fields like mathematical optimisations and theoretical studies. These technologies have the potential to be the base of the framework this research aims to create. Some of the early attempts to create software-based automata for different types of computation were based on and inspired by biology; John von Neumann (1966) (1963) describes the connection between the computers of the time and neurons as well as the need of software-based solutions for complex calculations. Some of his ideas can be described as emulation, although he himself uses the term black-box modelling to describe and show which parts can be made more abstract than what can be seen in nature. By demonstrating these as exceptions, the result of his work is closer to a simulation. What he created was the first of the major biologically inspired algorithms later named cellular automaton.

Cellular automata (CA) are machines that have independent cells doing the work, but these cells are simple ones and spatial orientation is very important. Stephen Wolfram (1983) defines CA as “Mathematical idealizations of physical

systems in which space and time are discrete, and physical quantities take on a finite set of discrete values". The strong spatiality and the related discreetness that would make CA a bad choice for general emulation can be seen in Christopher Langton's definition (Langton C. G., 1990): "Formally, a cellular automaton is a D-dimensional lattice with a finite automaton residing at each lattice site". The goal of CA is to do a very specific task, usually modelling-related. They are often used to model decentralised behaviour and are far less structured than the requirement for this research. It is reasonable to define CA as a machine in which each cell only has a local view and effect, which can be a major restriction if, for example, part of the simulation can be made more abstract by skipping some local causes and making an effect more global if it does not affect the results. Also the automaton has a strong spatial property that is rather rigid and therefore makes it harder to apply to some problems than one with more dynamic spatiality. It is not an absolute requirement but, mostly for simplicity cellular automata are discretely timed and synchronous. While this property makes it practicable to create accurate results as it enables precise control of the speed of each phenomenon, it makes the simulation potentially slow to execute as even the unaffected cells need to be updated in every iteration.

A more complex version of independently working cell-like behaviour is represented by multi-agent systems (MAS) (Wooldridge & Jennings, 1995). The agents are more suited for multi-scale simulations as they are asynchronous and can be hierarchically or spatially organised (e.g., agent-based modelling). There are examples of multi-scale biological simulations using multi-agent systems, for example the inflammation simulation (An, 2008) created with an agent-based modelling tool. There is no generally accepted definition of multi-agent systems, but certain properties are used in most definitions. Woolridge and Jennings's (1995) definition includes the following: autonomy, social ability, reactivity and pro-activeness. Pro-activeness here means that they do not just simply act in response to the environmental changes but also take the initiative by acting towards their goals. If emulating nature is the goal, these properties do indeed apply to biological cells. There is however another property of strict

locality in terms of sensing that, albeit logical and true to nature as no cell in a multicellular system has a truly global view (Panait & Luke, (2005), this constraint makes some possible future features and optimisations impossible or sub-optimal. In the interest of being as optimal as possible, which should be one of the main focuses of a generic emulation system, agents with global views will be allowed in my VLO framework. In every other sense, our system described is a multi-agent system.

On the strict simulation side, there are quite a few projects aiming at creating cell simulations as realistically as possible, like the E-Cell (Saito, et al., 2001) and the Virtual Cell (Loew & Schaff, 2001) projects. The concepts, challenges and solutions in these and similar projects (Takahashi, et al., 2002) are very different from the VLO due to the accuracy requirements (level of abstractness and accuracy).

Hardware engineering uses the concept of **embryonics**. The basic idea is to create the hardware version of molecules, which in turn can be coupled to become cells that can make an organism that has parts that can self-repair and self-replicate to some extent (Mange, Sipper, Stauffer, & Tempesti, 2000).This technology concentrates on a few parts of biology and does not imitate whole structures such as complex organisms made of complex organs; it rather resembles a multicellular system. Still it has some goals and tools similar to the present work, which we aim to realise virtually.

Among software tools there are also a few examples similar to the VLO. One of these is the language called little b (2008). The creation of this language was based on a starting idea almost identical to that of the present work: to create a model that can describe biology generally. The difference is that this language has goals that require it to be much more granular. It focuses on systems biology, rooted at the molecular level, turning it into something more like a simulation from our point of view (i.e. incorporates too many details). Little b also does not have the **middle-out** approach, it starts from the smallest possible scale, and this is rigidly enforced so crossing multiple scales is hard.

Aiming to emulate the mechanics of a complex organism makes it necessary for us to make our model much more abstract than little b.

Another tool that is formally somewhat similar is the multi-agent based Artificial life Framework (artificiallife.org, 2010). This is a Java framework that implements multi-threaded agents with its own internal mechanics (like messaging). While this framework is not based on biology and has very different goals, its details bear some resemblance to the initial goals of this research to create a C++ framework to simulate agent-like cells running in their own threads, although it differs too much to be of any use in this framework.

1.4 Methodology

Having reviewed the literature (see Chapter 2), I concluded that none of the existing approaches can achieve what our aims were. Therefore we have created a framework that helps users to create virtual living organisms. Since the goal is to emulate highly complex systems, the speed of execution is very important. My programmer background exposed me to many programming languages and tools, and my past experience with high throughput data handling and manipulations taught me that the highest level, easy to use general programming languages and tools rarely produce fast-executing programs, given their goals of general abstraction which makes them easy to use but less optimised for a specific task. These tools also often have limitations that can prevent them being used successfully or with acceptable performance for certain tasks. We view this as an unacceptable risk in long-term projects. This is why the chosen programming language for the project is one of the fastest general-purpose high-level programming languages, C++ (Stroustrup, 2000). This language also has the advantage of producing architecture independent code if used right, and this would enable users to use the framework in many different environments.

Using the same reasoning, a tool that enables multi-threaded programming (due to parallelism in nature) has to be chosen. Since the C++ language was created before multi-threaded programming was widely used, this technique is

not incorporated into it. The aim here is the same as in choosing the language, namely to choose a tool that gives the greatest control over threading but is easy to use and implement. There are many such tools widely used, including ones favoured by many scientists that enable rather effortless automations to convert single-threaded code to multi-threaded (e.g. OpenMP). While this makes working with them easier, the lack of fine control can mean that at some point in the future, there is no way of precisely controlling a cell or any intercellular mechanism. To make a choice that enables fine control and happens to be both future-proof and multi-platform, the chosen tool (Threading library) is the one that is to be included in the next C++ standard, the C++0x. This tool can be accessed at the moment as part of the much praised, high-quality boost libraries (www.boost.org). Since this library is a high quality and addresses many useful problems, and parts of it are often candidates for the newest standard plans, it is used extensively where its solutions give better performance than alternatives in- or outside the language.

The first task is to create a “basic skeleton” of the framework that allows the creation of a working VLO and should consist of object placing, navigation in the hierarchy, addressing of objects, communication and threading. After the basics are created, several example organisms need to be created to thoroughly examine the error-free operation of the framework. After modifying the code to be easily usable by minimising the necessary steps to create parts of a VLO, a complex example needs to be created that demonstrates the rapid prototyping capability of the framework.

1.5 Thesis organisation

The thesis consists of four main parts. After the introduction and literature review (Chapters 1 & 2), the programming framework (outlining a programming paradigm) is introduced and details various functionalities, properties and connections of the VLO. Since the research leading to this thesis had the creation of the VLO framework as its main outcome, this part is not just descriptive, but also has many discussions throughout that detail the design

choices as well as the possible alternatives where appropriate, detailing the advantages and disadvantages of all the options, the biological appropriateness and functional requirements that led to the decision of the final choice. The part about the framework consists of the main structure and parts of the VLO detailed in Chapter 3, the communication between cells in Chapter 4, the life cycle and all related functions detailed in Chapter 5.

With the knowledge of how a VLO works internally, the third main part has many examples ranging from of tissue level, through organ level and finally whole organisms. Chapter 6 details what can be called subsystems, which include tissues and organs and have functions that help or improve the functions of almost any VLO. Chapter 7 has two VLO examples, the first one being a sample system that is distributed with the framework that shows how to create a working VLO without any functional usefulness as a tutorial to any user. The second example shows one of the test VLOs used to evaluate and test several aspects of the framework during its creation, and later on served as the base for the cancer emulation. Chapter 8 introduces the cancer emulation used as the “real” demonstration for the technology. This part shows not only the technology, but also the biology behind it, and again it includes discussion about many parts and aspects of the emulation.

The fourth main part comprises Chapter 9, which has some additional discussions, but these are more related to the limits and possibilities of the technology as critical discussion and possible solutions as well as extensions of the framework that can have major benefits in the future. The final chapter (10) has some concluding remarks and identifies the main contributions to knowledge.

The full developer’s documentation of all the objects, variables and functions of the framework can be found in the attached CD.

2 Literature review

2.1 Concepts

The cooperation of biological sciences and computer science is not a new notion, in fact the field called “natural computing” (de Castro L. N., 2005) (de Castro L. N., 2006) encompasses most of what these two disciplines can do for each other. Natural computing has three main branches (de Castro L. N., 2005) (de Castro & Von Zuben, 2004):

- Simulation and emulation of biology
- Biologically inspired algorithms
- Computing with natural materials (de Castro L. N., 2007)

This chapter reviews the software aspect of this interdisciplinary field, which encompasses the first two branches of natural computing.

Simulation or emulation of biology tries to reproduce or mimic the processes and entities of nature within an artificial computing device having the main goal of understanding nature or predicting certain aspects of it. On the other hand, biologically inspired algorithms use certain knowledge of how nature works to solve complex problems of all kinds that cannot be efficiently solved by conventional means. But how can we characterise the difference between these two concepts when they both seem to utilise information and resources from computer science and biology? It is initially obvious that they have somewhat different goals. While biological simulations and emulations mainly benefit biological sciences by helping them gain more insights, biologically inspired algorithms are designed to solve general problems of any field (e.g.: they are heavily used in solving engineering problems). This leads to another major difference, since biological simulations often try to be as precise as possible to give accurate results that can be used instead of performing real world experiments, they need to be as detailed as possible, in other words as close to nature as possible (or, more to the point, as close as required by the task).

Biologically inspired algorithms get their inspiration from nature and need not to be detailed or accurate, but generic and optimal enough to fit a wide range of tasks, so they are far more abstract than the simulations.

There is a third field that is often taken as part of biological simulations/emulations, but given the main differences described above it falls right in the middle of the two main branches of natural computing, and this field is artificial life (Langton C. G., 1988). Artificial life tries to recreate the properties of what is considered a living being inside the software medium. These life forms are neither too detailed nor too abstract to fit comfortably into any branch, and are used to benefit both biology at a higher level of structural complexity (e.g.: ecological) and to perform general tasks (e.g.: robotics, computer gaming, etc.).

To summarize, the main directions of change from biological simulations/emulation through artificial life to biologically inspired algorithms the following two paths can be identified (Figure 1):

- Less biological relevance of the results, in essence taking them further from biology and making them more general
- Increased amount of abstraction, fewer biologically accurate details

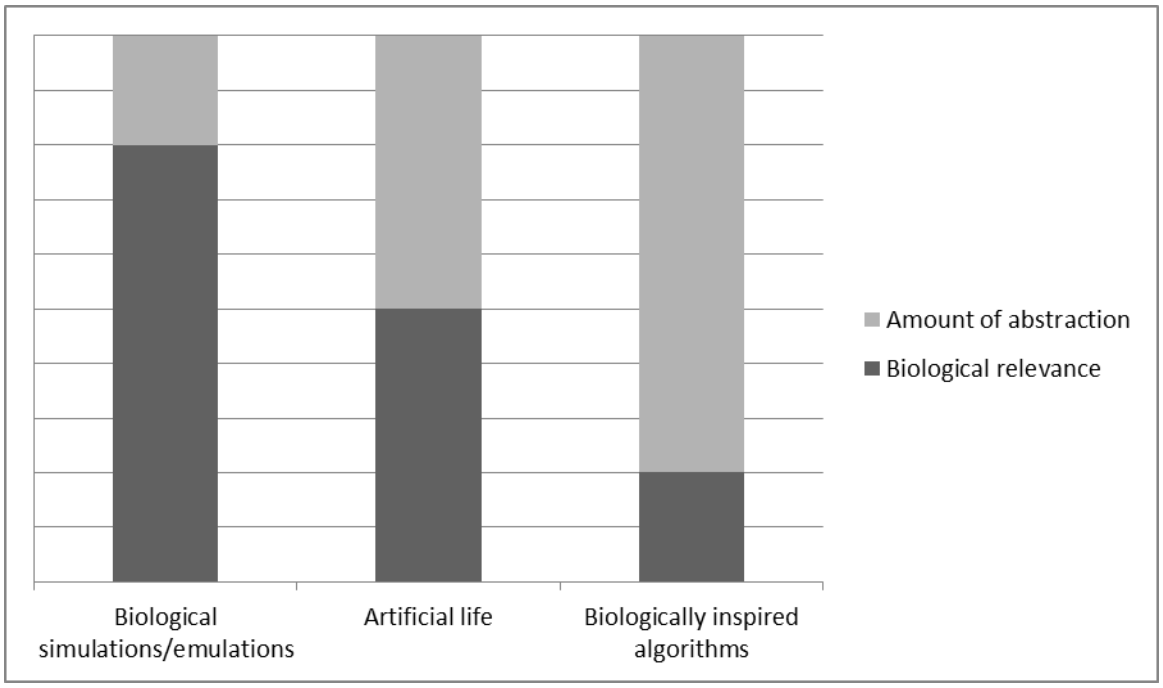


Figure 1 – Main directions of change from biological simulations/emulation through artificial life to biologically inspired algorithms

It is important to point out that there is no zero level of abstraction. We neither have enough information of the object of study (if it is obtainable at all) nor have the capacity to process the information to absolute precision (any information derived from an experiment is anyway only available to a finite level of precision). As such, the level of abstraction does not correlate with the quality (i.e., usefulness) of the model but is determined by the task at hand (Johnson, Goldman, & Gullick, 2004). The level of abstraction can be thought of as a tool of performance optimisation as increased abstraction can easily bring increased speed of the algorithm (i.e.: fewer details need to be handled), so in terms of speed it can be advantageous to find the maximum level of abstraction that produces the result of the required quality and precision. In reality, most of the algorithms used in natural computing are based on a highly simplified version of natural processes and entities (de Castro L. N., 2007).

Computer science uses abstraction heavily, and by doing so it has helped many other disciplines. This phenomenon influenced even the evolution of software technology. As one of the most often used programming paradigms, object oriented programming works by creating abstract objects that represent real ones in the problem's domain and creating processes that make these objects interact with each other and their environments creating complex behaviour that can be observed.

2.2 Biological simulations

In some cases traditional experimental and analytic techniques can be expensive, error-prone, hard to achieve or even impossible (e.g.: the sampling technique can destroy or change the result). In these cases biological simulations (or emulations) can replicate the natural processes or phenomena being studied in an easily controllable and observable environment. Even if traditional experimenting is possible, as a cost saving measure computer simulations can help test the feasibility of a hypothesis and determining what to exactly look for and what to handle carefully even before the experiments start (Johnson, Goldman, & Gullick, 2004). It can also produce an approximation of the expected results and can give insight to parts of the system that are not observable during the experiment (e.g.: by slowing down the simulation) (Johnson, Goldman, & Gullick, 2004). It can also help with more theoretical studies like how certain functions arise in biological organisms (Southern, et al., 2008). Being able to prove the sufficiency of a system in a hypothesis can guide a research process, just as proving the insufficiency of it can lead to the discovery of new components playing a critical role in the system (Johnson, Goldman, & Gullick, 2004).

Many industries use simulations to save R&D costs while others most notably the pharmaceutical industry, lag behind by still spending almost 25% of its revenue on heavily experimentally oriented drug development, which is almost double the typical amount spent on equivalent work in other high-tech industries (Models that take drugs, 2005) (Ghosh, Matsuoka, & Kitano, 2010). For the

pharmaceutical industry using simulation tools can make drug development and testing cheaper, faster and more efficient (Bugrim, Nikolskaya, & Nikolsky, 2004) (Cho, Labow, Reinhardt, Oostrum, & Peitsch, 2006) (Kumar, Hendriks, Janes, Graaf, & Lauffenburger, 2006) (Materi & Wishart, 2007).

A simulation always starts with a hypothesis of the workings of a complex system. This hypothesis is then represented in an abstract mathematical form, in a model. The model may have many parameters that can be used to fine tune it to match the real biological system, and these parameters with all their possible values define the parameter space. The models are then used with software tools that can solve the numerical equations of the model to create the simulations. These software tools are the simulators. These simulations can then be fine-tuned by moving in the parameter space to match some past experimental data via an iterative process.

Simplifications in a model are unavoidable, the kinds of which are dependent on the problem, the user, the expected output, the complexity of the system and the knowledge of the parameters in the parameter space (Materi & Wishart, 2007). The result is a simulation that only gives an approximation of what would be observable on the real system, but this information can give insight into the influencing factors of the original one (Southern, et al., 2008). Finding and understanding the minimal required system that produces a phenomenon can help to understand the evolutionary process that may optimise efficiency in a similar fashion (Johnson, Goldman, & Gullick, 2004).

The level of abstraction also defines the finest level of the real system represented by the model (abstraction can still be present at higher levels). In biological simulations this finest level characterises the system and can usually be classified as one of the following: subcellular, cellular, tissue, organ, organ system, organism, and environment. It can be even more finely grained by dividing, for example, the subcellular into quantum, molecular and macromolecular levels, although even then the genetic information does not exactly fit into any of those finer categories (Southern, et al., 2008). The enormous differences in size and timescale of phenomena in biology make it

very difficult to achieve higher accuracy in complex systems (Ridgway, Broderick, & Ellison, 2006).

The most often used models in biological (and scientific) modelling are based on ordinary differential equations (ODEs) (de Jong, 2002). There are many approaches of creating ODE models with different properties, strengths and weaknesses and it is often advantageous to combine them to create a more optimal system, like the combinations of discrete and continuous approaches (e.g., in spatiality (Ridgway, Broderick, & Ellison, 2006), (Sanford, Yip, White, & Parkinson, 2006), (Mallet, 2006)) or stochastic and deterministic approaches (Burrage, Tian, & Burrage, 2004).

2.2.1 Subcellular simulations

The subcellular level, which is also often called the mesoscopic scale (Southern, et al., 2008) here actually means everything smaller than the cell. This area is actively being researched, and many subcellular processes have already been simulated, like the actin-myosin dynamics (Negroni & Lascano, 1996) (Campbell, Razumova, Kirkpatrick, & Slinker, 2001). This scale has many complex processes that can be understood more easily with the help of computer simulations (Johnson, Goldman, & Gullick, 2004). It can be used for example to distinguish between hypotheses; for example Whalley at al. (2002) used it to determine if a protein moves freely within a cell or not.

One particular problem in this scale is the difficulty of observing certain phenomena without interfering with them through the act of observation. In some cases it is impractical or even impossible to observe quantitatively, just qualitatively (e.g., tagging techniques in protein/protein interactions (Hayes, Howard-Cofield, & Gullick, 2004) in which the tagging might itself change the nature of the phenomenon being observed). Such observations might be possible without this disadvantage using simulation tools (Johnson, Goldman, & Gullick, 2004).

Another use of simulations is to help create coherent hypotheses of individually known phenomena where the connection between them is not yet established. A simulation can easily be run with different parts added or left out, thus observing the change of global behaviour as a result of changing components. This can lead to better understanding of the interaction between parts in a complex system that might be hard to show in experiments. For example the use of neural networks and genetic algorithms can help to understand the connection between various proteins using data from many individual gene expression experiments (Keedwell, Narayanan, & Savic, 2002) (Keedwell & Narayanan, 2003).

According to Noble (2002), drug therapies affect proteins that work in a certain context in the body, but without knowing the protein's interaction at a higher level it is almost impossible to see which enzyme, receptor or transporter is relevant at the studied stage of the disease, and this can easily lead to side effects. This information cannot usually be practically obtained experimentally, but a feasible approach could be to create models and run biological simulations. Examples are modelling drug metabolism, the result of different drug concentrations or frequencies of administration to tumour, viral or immune responses (Bugrim, Nikolskaya, & Nikolsky, 2004) (Cho, Labow, Reinhardt, Oostrum, & Peitsch, 2006) (Kumar, Hendriks, Janes, Graaf, & Lauffenburger, 2006). There are also models and simulations based on network theories, like the ones researched by the LINK-Group in Budapest (Farkas, et al., 2011) (Antal, Böde, & Csermely, 2009).

Systems biology, working at this subcellular scale, creates lots of models from vast amounts of collected experimental data, and makes predictions from simulations verified by subsequent experiments, which constantly lead to the models being corrected to improve these predictions (Hood, 2003) (Ideker, Galitski, & Hood, 2001). This heavy use of models and simulations gave birth to a new discipline called computational systems biology (Kitano, 2002) (Materi & Wishart, 2007).

To help sharing and reuse of models in systems biology, a standardised language called Systems Biology Markup Language (http:// www.sbml.org) was created, and to help graphical representations to be in a common, standardised form, the Systems Biology Graphical Notation (http://www.sbgn.org) was created.

There are several models developed for more general use. Among them, the "E-Cell" (Tomita, et al., 1999) is regarded as the most detailed one (Johnson, Goldman, & Gullick, 2004). These more general models can be applied to many different simulations; for example one of the "Virtual Cell"'s uses could be to simulate calcium dynamics in a neuronal cell (Loew & Schaff, 2001).

There are many specialised model and simulation systems, like the Walk program (Lamb, 1996) that simulates the G-protein cascade using the random walk algorithm, and the StochSim simulation (Morton-Firth & Bray, 1998) that helps understanding of bacterial chemotaxis by examining temporal changes in a key cytoplasmic protein with a stochastic algorithm. Others simulate the stimulation and clustering behaviour of epidermal growth factor receptors on a cell's surface (Goldman, Gullick, & Johnson, 2004) or Boolean models of gene expression (Narayanan, Keedwell, & Olsson, 2002).

The base of the biological models can be of many kinds, but ODEs are popular, as in the "V-cell" (Slepchenko, Schaff, Macara, & Loew, 2003), the aforementioned "E-cell" (Yoichi Nakayama & Tomita, 2005) or simulation of the fission yeast cell cycle (Sveiczer, Tyson, & Novak, 2004), as a few examples of the finer-grained ones. Other highly mathematical models use for example partial differential equations or stochastic differential equations. There are also less mathematical methods; e.g., Petri nets (Pinney, Westhead, & McConkey, 2003), which have been used to create a qualitative model of apoptosis (Heiner, Koch, & Will, 2004); signal transduction pathways such as the mating pheromone response pathway in *Saccharomyces cerevisiae* (Sackmann, Heiner, & Koch, 2006) and the systematic analysis of metabolic disorders (Chen & Hofestadt, 2006).

2.2.2 Cellular simulations

Cells are considered to be the basic structural and functional unit of life (Southern, et al., 2008), or even ‘the building blocks of life’ (Alberts, et al., 2002). Cells are self-contained, isolated (by at least a semi-permeable membrane) and the right size to both encapsulate microscopic processes and affect macroscopic ones (Southern, et al., 2008). These properties make them a natural starting point for many simulations. Some example projects are simulations using cardiac ventricular cell models (Noble & Rudy, 2001) (Rudy & Silva, 2006). Since there are so many simulations at this scale, similarly to the subcellular one, a standardised mark-up language, the CellML was developed that allows sharing and reuse of models (Lloyd, Halstead, & Nielsen, 2004) (Schilstra, et al., 2006). An example project using this language is the COR software package (Garny, Kohl, & Noble, 2003), that can simulate 1 s of cardiac electrical activity in less than 1 ms.

Models at this scale include cellular automata (CA) and agent-based models (ABM). By default, CAs put cells on a two-dimensional lattice and use discrete time steps for updating the system; however there are variants for both asynchronous updates and less restrictive spatial orientation. On the other hand, ABMs do not require these limitations (but to create a practical and optimal simulation system, these are sometimes necessary (Materi & Wishart, 2007)). A few examples that use ABMs are: the simulation of bacterial chemotaxis (Emonet, Macal, North, Wickersham, & Cluzel, 2005), prediction of calcium-dependent pattern of wound closure in epithelial cell monolayers (Walker, Hill, Wood, Smallwood, & Southgate, 2004). Usage in pharmaceuticals involve optimising breast cancer vaccination protocols (Lollini, Motta, & Pappalardo, 2006) and simulating the innate immune response to predict outcomes of anticytokine treatments for systemic inflammatory response syndrome (SIRS)/multiple organ failure (MOF) (An, 2004).

2.2.3 Suborganism, supracellular simulations

2.2.3.1 Tissue level simulations

Tissue level simulations model the interactions between cells in the extra-cellular matrix (ECM) that give the cells structural stability (Southern, et al., 2008). Simulations at this level can be relatively fast if there is no need to interact with cell level simulations. Reviews of simulations of this level can be found in the book by Keener and Sneyd (1998) and Humphrey's (2003) paper. An overview of cancer modelling at this level can be found in the book by Preziosi (2003).

2.2.3.2 Organ level simulations

An organ is a discrete unit that usually has a main tissue and several sporadic tissues (Southern, et al., 2008). Organ systems are a group of organs that work together to achieve a common goal (like the respiratory system). Although there are several examples of simulations at this level (Shorten & Upreti, 2006) (Crampin, et al., 2004), the cardiac system is one of the areas most intensively simulated. An ODE model of this can be found in Noble's (2006) paper. Most of the times the simulation is not of the whole heart, but subunits of it, like ventricles or the atria (Southern, et al., 2008). Even these subunits have a long simulation time, it often takes days to simulate hundreds of milliseconds (Southern, et al., 2008). Anatomically accurate models have been developed of the hearts of dogs ((Nielsen, LeGrice, Smaill, & Hunter, 1991); (LeGrice, et al., 1995)), pigs (Stevens & Hunter, 2003), rabbits (Vetter & McCulloch, 1998) and humans from MRI (Watanabe, Sugiura, Kafuku, & Hisada, 2004) and CT scans (Xia, Huo, Wei, Liu, & Crozier, 2005). For a review of works in this field, see Kerckhoffs et al. (2006).

2.2.4 Organism and supraorganism level simulations

Organism level modelling mainly consists of the construction of a virtual physiological human (VPH) (Southern, et al., 2008), which is constructed with a bottom-up approach (Hunter & Borg, 2003).

There are simulations in which many organisms must be simulated. Usually in these simulations an individual is the basic unit without any lower level being simulated. This supraorganism level includes simulations of ecosystems and their environments.

Individual-based models (IBM) were developed in the 1980s for ecological models (Huston, DeAngelis, & Post, 1988) (Bousquet & Page, 2004). There were many studies that compared simulation results with mathematical models (McCauley, Wilson, & de Roos, 1993) (Wilson, de Roos, & McCauley, 1993) (Wilson W. , 1996). After gaining some acceptance, other simulations were created in the 1990s (Roese, Risenhoover, & Folse, 1991) (Silvert, 1993) (Derry, 1998) including forest models (Deutschman, Levin, Devine, & Buttel, 1997) models of human societies (Gilbert & Doran, 1994) and models of problem-solving skills of social insects (Deneubourg & Goss, 1989).

Several simulations used multi-agent systems (MAS) (comparable to cellular automata, see Chapter 2.4.2) in ecosystem management. They have been used for water (Lansing & Kremer, 1994), fishery (Bousquet, et al., 1993), park (Deadman & Gimblett, 1994), lake (Janssen & Carpenter, 1999), agricultural land (Dean, et al., 2000) (Balmann, 1997) and archaeological issues (Kohler & Carr, 1996).

There are also theoretical researches, such as the importance of the representation of agents in a multi-agent simulation (Dumont & Hill, 2001) or the relation between the stability of an ecosystem and its degree of connectivity (May, 1973) (Lindgren & Nordahl, 1994).

MAS have also been used in sociological research because of the common bottom-up approach (Epstein & Axtell, 1996) and are considered to be a good model to use for collective management of renewable resources (Bousquet, Barreteau, Page, Mullon, & Weber, 1999).

Not all ecological models use MAS or IBM, there are several other underlying algorithms, like cellular automata (CA, see Chapter 2.4.2) used in a model of

epidemics (Maniatty, Szymanski, & Caraco, 1993) and social behaviour of competing species (Galam, Chopard, & Droz, 2002), opinion dynamics (Galam, 2011) and predicting the spread of severe acute respiratory syndrome (SARS) by using past outbreak data and a stochastic metapopulation compartmental model (Riley, et al., 2003).

2.2.5 Multiscale simulations

Data used today in research comes from experimental observations, but the complexity requirements of observing multiple spatial or temporal scales restrict most observations to a single scale. However, for example, the spatial scales required to create a complete model of the human body range from 1 nm (proteins) to 1 m (human body), while the temporal scales range from 1 μs (ion channel gating) to 10^9 s (human lifespan) (Hunter & Borg, 2003). These problems arise in most biological disciplines, even computational systems biology struggles with creating modelling tools that can deal with this wide range of granularity (Materi & Wishart, 2007). Models of different scales in a multiscale simulation can help each other by setting up boundary or initial conditions based on their own results in some cases (Southern, et al., 2008). These multiscale models are still in their infancy (Southern, et al., 2008). There are several difficulties in multi scale simulations, for example the inaccuracies from averaging measurements over space and time (Rajesh & Sinha, 2008).

Cardiac simulations use multiphysics (electrophysiology, tissue mechanics, metabolism and circulation) and multiscale (cellular, tissue and organ) models (Southern, et al., 2008). The spatial scales cannot be easily uncoupled as there are two-way feedback mechanisms between them (Southern, et al., 2008).

A strongly coupled cardiac simulator (Nickerson, Smith, & Hunter, 2001) (Nickerson, Smith, & Hunter, 2005) used decoupling of parts of the multiphysics models (electrophysiological and mechanical) in each time step and advanced time using the explicit Euler method (Smith, Nickerson, Crampin, & Hunter, 2004). A comparable but much faster simulation was created by Usyk and McCulloch (2003). The simulator created by Wanabe et al. (2004) includes a

third physical process, the blood flow between the heart's chambers modelled as a Newtonian fluid. Kerckhoffs et al. (2005) created a simulation that eliminated the need for a cellular level model by coupling a tissue level eikonal-diffusion model to a nonlinear elastic model of cardiac mechanics (Kerckhoffs, et al., 2003), which makes membrane potential fully determined at the tissue level.

More wider scale and complex simulations might be able to determine whether the effects of events in a very small scale can affect much higher scales, as can be seen in projects like as the Physiome Project of the International Union of Physiological Sciences (IUPS); (Kohl, Noble, Winslow, & Hunter, 2000); (Hunter, Robbins, & Noble, 2002); (Hunter & Borg, 2003) that aims to construct a quantitative description of the physiological dynamics and functional behaviour of the intact organism (Bassingthwaighte, 1995).

Many of the multiscale approaches use cellular automata (CA) (see Chapter 2.4.2) and a closely related technology, agent based modelling. For example see the agent-based model simulating angiogenic sprout initialisation in response to VEGF (Bentley, Gerhardt, & Bates, 2007) or the CA based tumour expansion model (Kansal, Torquato, Chiocca, & Deisboeck, 2000) that help determine the role of random mutations in tumour subpopulations.

A three dimensional tumour model created by Zhang et al. (2006) simulates proliferation control based on the ODE-based cell cycle model of Tyson and Novak (Tyson & Novak, 2001). The role of the interactions between E-cadherin and beta-catenin in tumour invasion was studied with a multiscale model by Ramis-Conde et al. (2008). A simulation of the effects of blood flow on tumour based on CA was created by Alarcon et al. (2003).

The inflammation process is also studied with many multiscale simulations. A combination of an agent based model and a network flow model gave some insight into how inflammatory cell trafficking works in the microcirculation in the inflammatory process (Bailey, Thorne, & Peirce, 2007). Acute respiratory distress syndrome (ARDS) caused systemic inflammation was simulated by

combining an agent model of endothelial and inflammatory cell interactions (An, 2001) (An, 2004) and a hierarchical agent model of the organ luminal surface (An, 2008). This work of An (2008) is one of the few models using supra-cellular scales (Walker & Southgate, 2009).

Although multiscale approaches have been applied mostly to mammalian systems, there are several non-mammalian examples as well, like bacteria (Lees, Logan, & King, 2007), dictyostelium (Dallon, Jang, & Gomer, 2006), xenopus morphogenesis (Robertson, et al., 2007), central nervous system of the Drosophila embryo (Luthi, Chopard, Preiss, & Ramsden, 1998) and meristem development in plants (Stoma, et al., 2008).

Further reviews of multiscale modelling can be found by Schnell et al. (2007) and Bassingthwaighte et al., (2005).

2.3 Artificial life

According to the inventor of the term, “Artificial life is the study of man-made systems that exhibit behaviours characteristic of natural living systems. It complements the traditional biological sciences concerned with the analysis of living organisms by attempting to synthesize life-like behaviors within computers and other artificial media. By extending the empirical foundation upon which biology is based beyond the carbon-chain life that has evolved on Earth, Artificial life can contribute to theoretical biology by locating life-as-we-know-it within the larger picture of life-as-it-could-be” (Langton C. G., 1988)

As described by Langton, artificial life is about structure, organisation and function rather than the materials that build up an entity. If life can be accepted not to be chemical-based (much less carbon-based), but a collection of qualities and structural properties, then life can exist in other media made from any kind of substance. Life created from non-biological material (in this case inside the computer made of program code) is called Artificial life, or ALife. Even if the field is considered to have become established in the late 1980s, the main ideas were described much earlier in the 1940s by Schrödinger (1992).

There is no centrally accepted definition of life, it changes depending on the view of the discipline in which it is defined in (e.g.: biology or philosophy), and there are many definitions (Emmeche, 2000) (Margulis, Sagan, & Eldredge, 2000) (Rennard, 2004) (Schrödinger, 1992) (Sommerhoff, 1950). ALife research tries to assemble a set of required properties and create entities that demonstrate these (Farmer & Belin, 1991) (Ray, 1994), or provide relationships between biological and artificial agents (Keeley, 1997). One example is Farmer and Belin's (1991) list of eight properties defining life:

1. Life is a pattern in space-time, not a material object
2. Self-reproduction (at least to some degree)
3. Information storage of self-representation
4. Metabolism
5. Functional interaction with the environment
6. Independence of parts
7. Stability under perturbation and insensitivity to small changes
8. The ability to evolve (as a population)

Emergence ("properties at a certain level of organization which cannot be predicted from the properties found at lower levels" (Emmeche, Koppe, & Stjernfelt, 1997)) is a central phenomenon in ALife research (Emmeche, Koppe, & Stjernfelt, 1997) (Johnson S. , 2002). The centralised behaviour of a flock of birds can be achieved with only a small set of simple rules for every individual who only uses their own local perception of their dynamic environment, as described by Reynolds (1987) in a classic ALife work. To create flocking behaviour, each agent (or boid) must have three steering behaviours (Reynolds, 1999): separation, alignment and cohesion.

Artificial life research is used in arts and games (Maley, 1999), research of evolution and realisation of life (Maley, 1999) (O'Neill B. , 2003) (Ofria & Wilke, 2004) (Ray & Hart, 1998) (Rennard, 2004) (Wilke & Adami, 2002), linguistic

research (Christiansen & Kirby, 2003) (Kirby, 2002) and technological areas like robotics (Cazangi, Von Zuben, & Figueiredo, 2006) (Kube, Parker, Wang, & Zhang, 2004) (Nolfi & Floreano, 2002).

2.4 Biologically inspired algorithms

“In a broad sense, natural computing has taught us that any (model of a) natural phenomenon may be used as a basis for the development of novel algorithmic tools for problem solving[…]. The fruits of these explorations are continuously becoming new technological solutions and explanations to old and recent problems, and the full potential is far from being reached” (de Castro L. N., 2007).

Biologically inspired algorithms get their ideas from nature. Evolution managed to solve several difficult problems, and heavily optimised all solutions to make species tolerant and efficient. The method that these species used to overcome the difficult problems is analysed and generalised to be used with many other problems. This abstraction makes these algorithms the farthest from nature from all the technologies introduced in this chapter. Most of these algorithms either work with problems that can hardly be solved with other solutions (like artificial neural networks) or as general optimisation algorithms (like evolutionary algorithms or swarm intelligence algorithms). These solutions are used even in biological research, but their main uses are outside of biological sciences.

Some of the early attempts to create software based automata for different types of computation was based on and inspired by biology as John von Neumann (1966) (1963) describes the connection between the computers of the time and neurons as well as the need of software based solutions for complex calculations.

2.4.1 Artificial neural networks

The first mathematical model of a neuron was presented by McCulloch and Pitts (McCulloch & Pitts, 1943) which become the main element of the artificial neural

networks (ANNs) (Bishop, 1996) (Fausett, 1994) (Haykin, 1999) (Kohonen, 2000). As the name indicates, ANNs have similar features and performance characteristics to their natural counterparts. In principle they can be used to compute any function a computer can, but are best at clustering, classification, pattern recognition, and function approximation (de Castro L. N., 2007). Uses include knowledge extraction (Hruschka & Ebecken, 2006) (Jacobsson, 2005) and unification of language and cognition in evolving and integrated systems (Perlovsky, 2006).

The main features of ANNs are the following (Munakata, 1998):

- Topology: multi-layered, single-layered or recurrent. A multi-layered network has distinct layers where there is no connection between neurons inside a layer. A single-layered network is where there are no such layers. A recurrent network has the information flow in one way (feed-forward), but has at least one feedback loop.
- Data flow: recurrent or non-recurrent. In recurrent networks information can flow in any direction between layers, while in non-recurrent networks it can flow just in one direction.
- Types of input values: binary, continuous or sigmoid. These are the input values of the individual neurons, respectively (0,1), real numbers and (-1,1).
- Forms of activation: linear, step or sigmoid. This defines the behaviour of the neurons.

Some engineering problems are impossible or impracticable to find a perfect solution for. In many of these cases an acceptable solution is preferred, and ANNs can be a good solution for these problems. These algorithms have the following features to tackle these problems (Meireles, Almeida, & Simoes, 2003):

- Learning from training data used for system identification (Rumelhart, Widrow, & Lehr, 1994)
- Generalisation from new data after the training phase (Jung & Hsia, 1998)
- Mapping of nonlinearity (Payeur, Le-Huy, & Gosselin, 1995)
- Parallel processing
- Applicability to multivariable systems
- Black-box approach

The first industrial use of ANNs was the echo cancellation of telephone lines (Widrow & Hoff, 1960). There are four main categories of industrial applications of ANNs (Meireles, Almeida, & Simoes, 2003):

- Modelling and Identification
- Optimization and Classification
- Process Control
- Pattern Recognition

ANNs have also been used in energy systems (Kalogirou S. , 1999) because of their fault tolerance, robustness and noise immunity (Rumelhart, Hinton, & Williams, 1986). Some examples are the modelling of the start-up (Kalogirou, Neocleous, & Schizas, 1996) and prediction of the steam production of a solar steam generator Artificial neural networks for the estimation of the performance of a parabolic trough collector steam generation system, (Kalogirou, Neocleous, & Schizas, 1997) and load forecasting (Czernichow, Germond, Dorizzi, & Caire, 1995), (Khotanzad, Abaye, & Maratukulam, 1995).

They have also been applied to polymer composites (Zhang & Friedrich, 2003) to predict wear volume (Velten, Reinicke, & Friedrich, 2000), mechanical properties (Zhang, Klein, & Friedrich, 2002) and determine the

control parameters in manufacturing (Allan, Yang, Fotheringham, & Mather, 2001) amongst others.

2.4.2 Cellular automata

Cellular automata (CA) are made of a large number of computing cells arranged into a lattice. These cells are almost identical (usually with identical programs), but can have separate states and can sense and interact with their local environment (neighbours). Stephen Wolfram (1983) defines CA as "Mathematical idealizations of physical systems in which space and time are discrete, and physical quantities take on a finite set of discrete values". Most CA update their cells synchronously with discrete time steps. This strong spatiality and the related discreetness can be seen in Christopher Langton's definition (Langton C. G., 1990): "Formally, a cellular automaton is a D-dimensional lattice with a finite automaton residing at each lattice site". In most CA cells remain stationary, but in a variant known as dynamic cellular automata (also known as agent based modelling) (Wishart, Yang, Arndt, Tang, & Cruz, 2005) cells are able to move (Wishart, Yang, Arndt, Tang, & Cruz, 2005). In contrast to ODEs, CA algorithms are robust and more easily scaled both spatially and temporally (Materi & Wishart, 2007). To create a CA that can perform computations that require global coordination, Mitchell et al. (1996) applied genetic algorithms (see Chapter 2.4.4.1) to the design of CA.

CA based simulations are also used in biological research. They have been used to simulate basic enzyme kinetics (Wishart, Yang, Arndt, Tang, & Cruz, 2005) (Kier L. B., Cheng, Testa, & Carrupt, 1996) oscillatory gene circuits (Wishart, Yang, Arndt, Tang, & Cruz, 2005), myxobacterial aggregation (Sozinova, Jiang, Kaiser, & Alber, 2005), predator–prey relationships (Kondoh, 2003), drug release in bioerodible devices (Zygourakis & Markenscoff, 1996), lipophilic drug diffusion (Wishart, Yang, Arndt, Tang, & Cruz, 2005) (Kier L. B., Cheng, Testa, & Carrupt, 1997), drug-carrying micelle formation (Kier L. B., Cheng, Testa, & Carrupt, 1996), progression of HIV/AIDS and HIV treatment strategies (Zorzenon dos Santos & Coutinho, 2001) (Peer, Shah, & Khan, 2004)

and by mimicking certain properties of viral infections reproducing the three-phase pattern commonly found in patient T-cell counts and viral loads (Zorzenon dos Santos & Coutinho, 2001) (Materi & Wishart, 2007). Using L-systems (Room, Hanan, & Prusinkiewicz, 1996), CA can recreate complex natural patterns like seashells and forest ecosystems.

2.4.3 Artificial immune systems

Artificial immune systems (AIS) are adaptive systems inspired by theoretical and experimental immunology. The natural immune system has three layers, the anatomic barrier, the innate and the adaptive immunity. The latter two are interlinked and work together (Abbas & Lichtman, 2000). The immune system can be seen as a parallel and distributed information processing system with partially decentralised control that performs feature extraction, signalling, learning, associative memory retrieval and combinatorial tasks (Dasgupta D. , 2006). These properties make it appealing as a software modelling target.

There are five main types of general-purpose AIS algorithms in the literature (de Castro L. N., 2007), and they can be separated into two groups: population based that does not take into account the immune network, and network based inspired by the network theory of the immune system proposed by Jerne in the 70's (Jerne, 1974).

The main types of algorithms are:

• Bone marrow: generating populations of immune cells and molecules (e.g., (Hightower, Forrest, & Perelson, 1995) (Oprea & Forrest, 1998) (Perelson, Hightower, & Forrest, 1996)).

• Negative selection: defining a set of detectors for anomaly detection (e.g., (Forrest, Perelson, Allen, & Cherukuri, 1994) (González & Dasgupta, 2003) (Hofmeyr & Forrest, 2000)).

• Clonal selection: used to generate repertoires of immune cells driven by antigens. It regulates the expansion, genetic variation, and selection of attribute

strings (Cutello & Nicosia, 2004) (de Castro & Von Zuben, 2002) (Forrest, Javornik, Smith, & Perelson, 1993) (Kelsey & Timmis, 2003).

• Continuous immune network models: used to simulate dynamic immune networks in continuous environments (e.g., (Farmer, Packard, & Perelson, 1986) (Varela & Coutinho, 1991)).

• Discrete immune network models: used to simulate dynamic immune networks in discrete environments (e.g., (de Castro & Von Zuben, 2001) (Galeano, Veloza-Suan, & González, 2005) (Neal, 2003) (Timmis, Neal, & Hunt, 2000)).

There are hybrid versions as well, like the combination of clonal selection and immune networks by Wierzchoń & Kużelewska (2002).

AIS algorithms have many applications, like anomaly detection (Dasgupta & Forrest, 1996), (Dasgupta D. , 1996), pattern recognition (Cao & Dasgupta, 2003), data mining (Timmis, Neal, & Knight, 2002), adaptive control (Krishnakumar & Neidhoefer, 1999), fault detection (Bradley & Tyrrell, 2000), (Dasgupta, KrishnaKumar, Wong, & Berry, 2004) and diagnosis (Bersini & Varela, 1991).

AIS have been used heavily for intrusion detection, because of the similar use of its natural counterpart. There are three major branches of these algorithms (Kim, et al., 2007):

- Methods that use conventional algorithms inspired by the immune system, like the virus detector by IBM (Kephart, 1994)
- Algorithms based on negative selection (Forrest, Perelson, Allen, & Cherukuri, 1994) (Somayaji, Hofmeyr, & Forrest, 1997)
- Algorithms using danger theory (Matzinger, 1994)

There are also some newer methods like aiNET (De Castro & Von Zuben, 2000) and immunocomputing (Melnikov & Tarakanov, 2003).

AIS algorithms can also work to protect not only a single computer, but detecting misbehaving nodes in mobile ad-hoc networks An artificial immune system approach to misbehavior detection in mobile ad-hoc networks, (Le Boudec & Sarafijanovic, 2003) (Le Boudec & Sarafijanovic, 2004) (Sarafijanovic & Le Boudec, 2003) or malicious nodes in peer-to-peer networks (Trapnell, 2005).

2.4.3.1 Artificial immune systems and artificial neural networks

Both the immune system and the nervous system have similar functions in biology, namely recognition and categorisation (Dasgupta D. , 1997). There are differences of course, the immune system has long lasting, self-organising memory, and aims for diversification instead of converging to any optima (Frank, 1996). At the level of system behaviour, these systems are very similar, but they are different at their building-block level (Hoffmann, 1986). For example the lymphocyte cells of the immune system float freely in the blood and lymph, unlike neurons that are spatially fixed (Dasgupta D. , 1997).

There are many similarities as well, like the use of variable rate of cell division and programmed cell death for dynamic resource allocation, or the decentralised recollection of previously learnt information. The immune system has even been called the “second brain” because it can store information gained with past experiences and generate new responses to new patterns (Rowe, 1994). Similar characteristics to the associative memory of the Hopfield networks (Fu, 1994) can be found in immunological models (Hunt & Cooke, 1996) too.

2.4.4 Optimisation algorithms

2.4.4.1 Evolutionary algorithms

Evolutionary algorithms (EAs), as their name suggests, are loosely inspired by the method nature uses to make species more resilient and optimal, evolution. Note that real biological evolution is open-ended (i.e. with no stopping criterion), whereas its artificial counterpart is almost inevitably used to solve a specific

predefined problem. EAs are based on an iteration of selection (i.e. selecting a subset of the population as a base for the next generation), recombination (i.e. creating the individuals of the next generation from a combination of individuals from the current generation), and mutation (i.e. stochastic changes of some properties of the newly created individual) on possible solution representing entities. The goal is written as a function that can assign a fitness number to each solution, thus evaluating it. There are several distinct versions of evolutionary algorithms, depending on what is represented by a solution and in turn how the main iteration works. There are also many kinds of subversions of those depending on the problem they solve. The main versions are:

- Genetic algorithm (GA) (Goldberg, 1989) (Mitchell M. , 1996): possible solutions' phenotypes (observable characteristics encoded in the genotype) are typically encoded as binary strings; most versions use crossover, and mutation for creation of the next generation
- Genetic programming (GP) (Koza, 1992) (Koza, 1994): solutions are possible programs generating the desired individuals; typically they use crossover and mutation
- Evolutionary programming (EP) (Bäck, Fogel, & Michalewicz, 2000) (Bäck, Fogel, & Michalewicz, 2000) (Fogel, 1998): phenotypes represent distinct species, technically similar to genetic programming, but only evolves parameters of the system to be optimised; only uses mutation for creation of next generation, there is no crossover; typical version use stochastic tournament for selection
- Evolution strategies (ES) (Beyer, 2001) (Schwefel, 1965): initially used for parameter optimisation; phenotypes consist of real numbers representing individuals; most versions use mutation for creation of next generation (there is no crossover); typically use deterministic selection

Evolutionary algorithms are used in many disciplines as generic stochastic optimization algorithms, for example art and music composition (Bentley P. J., 1999) (Bentley & Corne, 2001), electronics (Zebulum, Pacheco, & Vellasco, 2001), language (O'Neill & Ryan, 2003), robotics (Nolfi & Floreano, 2000), engineering (Dasgupta & Michalewicz, 1997) (Farnsworth, Benkhelifa, Tiwari, & Zhu, 2010) (Farnsworth, Benkhelifa, Tiwari, & Zhu, 2010), data mining and knowledge discovery (Freitas & Rozenberg, 2002), industry (Karr & Freeman, 1998), signal processing (Fogel, 2000), and it can also be applied to dynamic (Arnold & Beyer, 2006), multi-modal, multi-objective (Farina, Deb, & Amato, 2004) and constrained (Venkatraman & Yen, 2005) optimization (e.g., data mining (Kushchu, 2005), games (Yannakakis, 2005), arts and music (Corne & Bentley, 2001)

2.4.4.2 Swarm intelligence

Swarm intelligence (SI) originally referred to cellular robotic systems where simple agent can have only local interactions with their environment (Beni, 1988) (Beni & Wang, 1989). White and Pagurek (1998) define SI as "a property of systems of unintelligent agents of limited individual capabilities exhibiting collectively intelligent behavior". The main two versions of SI are particle swarm optimisation based on the ability of human societies to process knowledge and ant-colony optimisation based on social insects.

2.4.4.2.1 Particle swarm optimisation

Particle swarm (PSO) optimisation algorithms, which are based on human social influence and cognition (Kennedy, 2004) mimic the way human societies can process knowledge (Kennedy, 1997) (Kennedy, 2004). They be used to optimise nonlinear functions (Kennedy & Eberhart, 1995), and are used, for example, in human tremor analysis, milling optimization, ingredient mix optimization, reactive power and voltage control, battery pack state-of-charge estimation, and improvised music composition (Engelbrecht, 2006) (Kennedy, Eberhart, & Shi, 2001). It can also find solutions to many kinds of optimisation problems, including dynamic (Blackwell, 2003), constrained (Coath &

Halgamuge, 2003), combinatorial (Pang, Wang, Zhou, & Dong, 2004), multi-objective (Zhang, et al., 2003) (Zheng, Ma, Zhang, & Qian, 2003) and niche (Brits, Engelbrecht, & van den Bergh, 2003) optimisations. There are also many hybrid (Naka, Genji, Yura, & Fukuyama, 2003) versions combined with elements of GP (Poli, Chio, & Langdon, 2005) and EA (Higashi & Iba, 2003).

2.4.4.2.2 Ant-colony optimisations

Ant-colony optimisation (ACO) was inspired by the foraging behaviour of ant colonies (Dorigo, Di Caro, & Gambardella, 1999). Although it works inherently with discrete search spaces, there are continuous versions available (Bilchev & Parmee, 1995). Bonabeau et al. (Bonabeau, Dorigo, & Théraulaz, 2000) states that ACO is the best available heuristic for the sequential ordering problem and it is also used for many other problems, like the travelling salesman problem, network and vehicle routing, machine scheduling and frequency assignment. Some other uses of natural massively parallel systems can be found in (Resnick, 1994). There are many good surveys of ACO, including (Bonabeau & Théraulaz, 2000) (Bonabeau, Dorigo, & Théraulaz, 1999) (Dorigo & Di Caro, 1999) (Dorigo & Stützle, 2004) (Dorigo, Di Caro, & Gambardella, 1999) (Engelbrecht, 2006). Some of the more successful variants of ACO are the Ant System (Dorigo, Maniezzo, & Colorni, 1996), the Max-Min Ant System (Stützle & Hoos, 2000) and the Hypercube framework (Blum & Dorigo, 2004) (de Castro L. N., 2007).

2.5 Conclusion: moving between main concepts

As can be seen from the examples, in some cases it is hard to classify a research subject. Sometimes organ level simulations include cellular processes and in some cases a cellular simulation uses cellular automata or a swarm intelligence algorithm. But does this mean that these researches can be classified as being in between these main branches? The answer is no, since even if a simulation uses biologically inspired algorithms, it remains a simulation and its sole purpose is defined by this fact. It also uses software tools but most people do not mainly associate software technology with it. Even so, a certain

level of movement between concepts would be beneficial. If someone wants to develop a biologically inspired algorithm or a new version of artificial life, they will almost certainly have to look for the exact biological mechanism that will act as a model for their work. However most of the biological simulations use either a custom built software or in the best case a framework that has been designed for a certain type of simulation. These frameworks however do not support much abstraction, so simulations created with them are hard to be of any use directly in either AL or general algorithm creation. It is almost the same in the other direction.

Biologically inspired algorithms can be a great tool to accomplish parts of even a biological simulation, but can hardly be transformed back into a simulation easily, even if the greater abstraction would make mechanisms transparent enough to lead to a discovery into how a related biological mechanism could work. If there were a framework that was precise enough if needed to accomplish accurate biological simulations, but each part of the simulation could be easily made less detailed and more abstract without affecting the rest of the mechanism then this would help transform simulations into real inspirations for algorithms, and theoretical ideas derived from AL to working simulations as proof. Such a tool would help many researchers in all of the fields mentioned in this review. One such tool is outlined in Bándi and Ramsden (2010) and detailed in this thesis.

3 Basic structure of the virtual living organism

At the time of writing, the virtual living organism (VLO) has four main structural levels: cells, tissues, organs and organisms. This can be easily extended in the future. All structural levels have their own uses: the cells do all the physical work in the system, the tissues holds cells that work closely together, the organs hold tissues that together can create a coherent function within a system and the organism consists of every organ that may be needed during its lifetime. This shows that only the virtual cells are the workforce, all the other levels above this in the hierarchy are created as structural levels and are actually only containers with added functionalities that support cells. This chapter describes in detail the properties, functions and parts of these levels.

3.1 Cell

The best way of creating a digital counterpart for the components of biological life forms is to look at the definition of the original biological unit. The Merriam-Webster dictionary defines the cell as:

“a small usually microscopic mass of protoplasm bounded externally by a semipermeable membrane, usually including one or more nuclei and various nonliving products, capable alone or interacting with other cells of performing all the fundamental functions of life, and forming the smallest structural unit of living matter capable of functioning independently”.

Of course almost every definition will include the original biochemical components that the unit in the definition is made of, and this is what needs to be changed in their digital equivalents, or at least something close to that. Even so, the original biological makeup of the unit can act as an inspiration. In this case, the smallness is emphasised more than once, so the digital cell should be a small part of the system. The semipermeable membrane suggests that it should be one whole unit, like an object in object-oriented programming, but should be allowed to have inputs and outputs. The nuclei and non-living parts suggest that it should have both active parts and inactive resources; this also

fits the definition of an object. Capable alone or interacting with other cells: since it is supposed to be small, living alone can mean a simple single cell organism, while working together can mean they can perform complex operations. It should also be the smallest unit that can live independently, so it should conform to the definition of living, and should be the smallest part that does this.

The definition of life according to the Merriam-Webster dictionary is:

“a state of living characterized by capacity for metabolism, growth, reaction to stimuli, and reproduction”

This means that the cell (and every other part that has the “living” property) should have: metabolism which in turn means the function of storing and processing of resources; growth which is a property that can have many different meanings depending on the actual situation; reaction to stimuli which is a form of adaptation; and reproduction (note that most mules are infertile and still definitely living creatures, so reproduction does not have to apply to every level).

There are also definitions of life in other sources that include the goal of life as self-preservation which is a reasonable assumption in view of the evolutionary process that seems to drive biology.

With these definitions, required properties of the digital version of the cell are outlined. The cells inside the VLO, as described in this document, meet all of these requirements. From this point on, the word cell will mean the virtual cell, unless it is specifically expressed that the biological cell is meant.

The cells are the workforce of the VLO; these entities are the only parts of the system that can execute any kind of code. In software, virtual cells are objects coupled with a thread (Hyde, 1999) (an individual line of execution in parallel with all other parts of the system) inside the program. In other words, threads are agents inside a multi-agent system. These cells have independent lives and work in a concurrent fashion.

The usual approach of multi-threading (Hyde, 1999) (parallel computing) in computer science is to create not many more threads than there are processor cores inside the system, as the maximum number of physically concurrent operations is limited by this property. The number of threads may vary from program to program, may even be affected by the operating system type, architecture or programming language standards, but usually is determined by the number of different tasks a program has to do at the same time (e.g.: user interface, networking, database operations, etc.). Having many more threads than cores will result in decreased speed of execution as the operating system has to swap thread data more often.

The reason why a one thread per cell approach was taken is that this is the actual way of nature; every cell can work independently (while being part of a bigger process) and concurrently. The speed penalty can be decreased later in many ways, depending on the usage. A thread pooling system (Hyde, 1999) (many virtual threads sharing much less physical ones) can keep the thread number at a lower level if needed, although sleeping threads (virtual threads not being executed at the moment but being kept in memory) do not slow the system down. There is also the possibility of using distributed systems or executing the emulation on a computer with many more processors or even a graphical processing unit (GPU) (see Chapter 9.5 for details). The results of this threading system is that a VLO program has the same number of threads as cells, plus one more thread that started the creation of the VLO.

Each cell is also an object that is inherited from the Cell superclass that has all the default functionalities inside that a cell could need during its lifetime. These functions include the handling of the link to the “parent” tissue the cell is contained in, thread handling functions that help in creating and destroying the cell, querying functions that return properties of the cell (like being dead), command functions that order the cell to die or prepare for death and job handling functions that help in communication. For further information on jobs see Chapter 4. The two main functions of the cell are the cell cycle that does

the real job of the cell and the maintenance function that tells the cell, amongst other things, when to divide or die.

The way every cell works is defined in the cell cycle (Figure 2). The cycle described here is a suggested structure, not mandatory. The cycle starts with the cell setting up its default values and doing any setup needed before running. After this starts a loop during which it waits for a job that specifies a task, and when it receives one the cell processes it. Each cell has a special purpose in "life", which is suggested to be a small particular task that correlates with the limited functionality of an individual biological cell. Of course if required, each cell can be as complex as the programmer makes it. After processing the job, the maintenance function is invoked to see if the number of jobs in the queue indicates, the cell should die that or self-replicate. When the maintenance is done, the cell waits for any new job and starts the loop all over again. There is a special kind of job for all cell types that tells the cell to die. If this is received, the loop is broken and the cell enters the shutdown phase. This may include any custom processes needed for the particular cell type, plus the emission of a signal that tells the system that this cell is dead and ready for its resources to be freed.

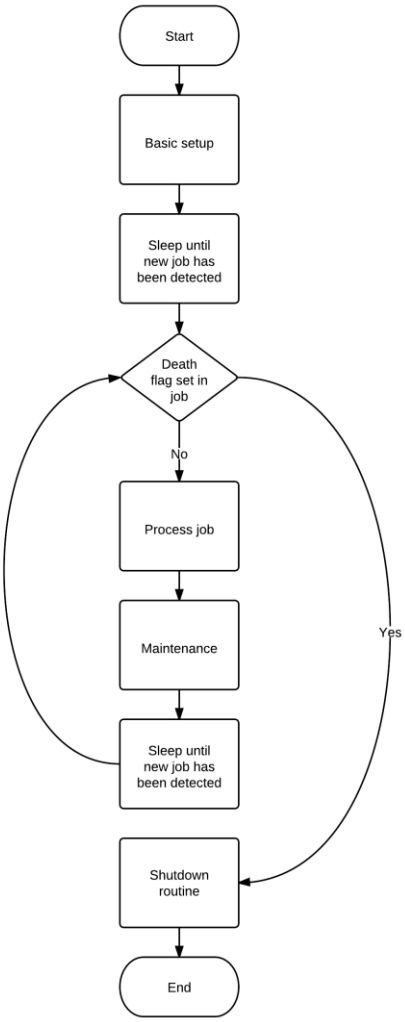


Figure 2 – Default suggested cell program structure including the cell cycle.

The creation of a cell is a complex process, and only the part that involves just the cell will be detailed here. For the other parts of this process in detail, see Chapter 5.2. Cells need three kinds of information to work, their internal variables (e.g., cell type ID, current state, optimised values for work, etc.); their place inside the organism which mainly consists of the tissue they are in; and

their thread. After all of these are set up and assigned properly, the cell thread can be started and the cell immediately starts its cycle as described above.

Cell death is an equally complex task; sub processes related to other structural levels are detailed in Chapter 5.3. The sub process that only involves the cell consists of the detection of the death signal, setting death markers (internal variables) inside the cell, signalling the event to the cell's environment so the remains of the cell can be processed and finally terminating the thread of the cell. Death signalling is a two-step process at the end of the organism's life. The first is a warning called the shutdown signal that tells the cells not to start processing new jobs and start preparing for death, the second signal is called the kill signal and this orders the cells to die immediately. After this signal the cells have a limited amount of time to die and then they are forcibly killed. If cell death was caused by optimisation (i.e. the cells decided to lower their numbers), the death signalling process is not used obviously (cells die without an external signal); the cell goes to the end of its main loop voluntarily and dies.

The before mentioned optimisation process is one of the great advantages of nature and its virtual counterpart compared to artificial systems. As evolution drove nature to be more and more optimised, the multi-cellular structure became dominant in more complex organisms. This structure helps the virtual system to create a dynamic load balancing by adjusting the number of cells of each type to share the workload. Each new cell helps in processing the jobs of its type a bit faster, but also consumes more resources, so self-replication must be used optimally. It is just like in nature where a bigger mass can give an advantage to an animal, but comes with greater nutritional requirements. In the VLO, cell replication is determined by the number of jobs inside the job queue of the cell type in question. If this queue keeps increasing over time, it means that the cells cannot cope with the amount of work incoming and this causes a performance bottleneck. If the number of this cell type increases, it means that the percentage of this cell type is also increased compared to the whole organism. The result is that these cells will be executed more by the processor cores and their work will be processed more than others'. The downside is that

it will require extra memory and overhead increased by switching between more cell threads. It is also counterproductive for all the cell types to increase because then not one will be increased compared to the others. So a self-balancing process had to be devised.

The self-replication process has two parts. The first is the work of a subsystem that reports the total number of cells inside the organism, which gives a picture of how saturated the system is, and helps in the decision whether to self-replicate or not. This is similar to quorum sensing (Waters & Bassler, 2005) in nature, where a group of cells can detect the density of their local population and this can help in their decision to self-replicate or not. Details of this subsystem can be found in Chapter 6. The second part is the aforementioned maintenance function inside the cell's main loop.

The maintenance function, just like the cell's main loop, is a sample function that should be adequate for any cell but can be changed if a cell type has special needs. As part of the process, some maintenance related variables are stored inside the cell and some inside the tissue in the form of a shared variable across cells of the same type. This is somewhat similar to the environment that all cells are aware of, like pressure or the concentration of a chemical. The maintenance function first checks the property of the cell type that describes if maintenance should be enabled for this type or not. This is needed because there are certain types of cells that only should have one instance. Such types include the **DivineCell**, the logger cell and the cell number synchronisation cells, all of which will be described in later chapters.

Shared variables inside the tissue are in what is called a critical section. This means that only one cell can modify this value at a time, like a molecule that travels from cell to cell and those cells change it from time to time through chemical reactions. This section prohibits multiple threads accessing the data at the same time, which could cause data inconsistencies and thus errors. The main idea is that only one cell of the same type can be in self-replication mode when it determines if it should replicate, die or return to doing its job. To avoid hundreds of cells trying to enter the critical section unnecessarily, there is a

timer that tells the cells when they should bother checking if the section is free. This timer increases the time between such checks with the number of cells so ultimately minimising the number of failed attempts. There is also a shared timer that says how often a cell inside the critical section should check if a change in cell numbers is needed. This is also increased by failed attempts and decreased by successful checks and cell number modifications. If the cell detects that it is time to check if the number of cells of its type should change, it compares the current number of jobs inside the job queue with the stored last number of jobs at the last cell number modification; and, if the difference is significant, it dies if there are much fewer jobs and self-replicates if there are many more new jobs. The amount the cell checks is essentially the change in the amount of jobs waiting. This is the value that should be checked regularly as it indicates how the current number of cells can cope with the workload they get. When the cell self-replicates, it does this many times to be optimal (quicker response for the workload increase), but always checks the current number of cells inside the organism so as not to reach the architecture limits. For more details on this process, see Chapter 6.2.

At the time of writing the self-replication algorithm, there were several options to choose from. The problem is that cells should be decentralised and should be able to handle the optimisation of their numbers collectively. If all the cells looked at the current conditions and based on that they initiated self-replication, then it could happen that many of these cells detect the need of, for example a 10% increase in numbers, but in the time the first finishes self-replication, many more than 10% of the cells may have initiated the process, and this would of course result in a much larger increase in their collective numbers. The current system emulates a hypothetical synchronisation mechanism that is initiated by a replication inhibitor represented by the critical section the cell enters at the beginning of the maintenance process, and ends with another signal that nullifies the inhibition. The other common alternative for these sorts of problems is a centralised solution. In that solution either a dedicated copy of the cells has the task of not doing its designed function but to watch for the cell numbers and number requirements this solution would create cells that are unneeded most of

the time and would make the convenient and optimal solution of being activated by incoming work harder to use. It would also make the cells a lot less like equals, which is not the case in nature most of the time. Another solution would be to have another type of cell checking for the need of cell number increase from time to time. An optimal choice in this case would be to have a dedicated support tissue for this task (because it is a supportive task), but in that case this tissue would have to look into other tissue's statuses which translated into physical terms could mean that it has knowledge of all the other tissues in the organ, even if they are not adjacent, and this would also be very inaccurate. The third option would be to have undifferentiated blank cells move around the organism and they could enter into any population of cells if needed. This solution however would be much slower in responding to increased workload than the current one. It would also not be able to decrease the number of cells. Having these cells move around most of the time would also mean that they are very much active, which is something that needs to be avoided if possible as it would involve computational work without any good reason and could slow down execution speed considerably. And the best reason not to use any outside source in the self-replication process is that the optimal number of cells could be different in each cell type. Not only are there cells that should functionally be singletons, but past experience could also teach cells to find an optimal number to cope with regularly fluctuating workload. For example a workload that resembles a sine function could be best processed with cell numbers that could process the average of the workload, if the length of a period is short enough compared to the job processing speed for the cells to work all the time and finish a period by the time the next one comes. This would make it worthwhile for the cells to learn workload history and adapt to it. The result is that no outside source could do this task as optimally as the cells themselves without extensive knowledge of the cell type in question. And there should be no cell type that has to know all the other types in detail.

Although it will be detailed later in Chapter 4, job handling functions are integral parts of all the cells, so they are worth mentioning here. Some of these functions are related to death, these are the **shutdown order** and **kill order**

functions. These functions return a job object that only has one of these orders inside. Others are the functions that create the job for the cell type in question, this is called the **jobOrder** function. This function is what is termed a static function, meaning that it can be called with a class (object type) instead of an actual object. So if a cell wants to create a job for **cellTypeA** that consists of a message, it calls **cellTypeA**.**jobOrder**(message). A small technical detail worth mentioning here is that this method of using static functions helps the creator of VLOs a great deal because most development environments look up the needed parameters of function while the programmer types in the function names, and offers a list of them. So when the programmer types in "**cellTypeA**.**jobOrder**(" the environment will list the parameters, in this instance it will display "(string message)".

3.2 Tissue

The Merriam-Webster dictionary defines tissue as:

"an aggregate of cells usually of a particular kind together with their intercellular substance that form one of the structural materials of a plant or an animal and that in animals include connective tissue, epithelium, muscle tissue, and nerve tissue".

This definition shows that tissue is a special kind of aggregate of cells, so is a structural entity that also includes the intercellular substance that can be found around cells. The second part of the definition is similar to the bio-chemical part of the cell's definition in the sense that it relies on nature, on what has been seen so far, and not particularly what might be there beyond that. The concrete examples here can be examples also in any virtual organisms, but should not be included in their definitions. From this point on, the word tissue will mean the virtual tissue, unless it is specifically expressed that the biological tissue is meant.

As will be described later, every hierarchical level of the VLO's system above cells is a logical level, that is, it is without a thread and only have "services" and

shared data in them. Services can be thought of as such callable functions that implement the particular uses that can be best achieved with the use of the hierarchical level in question. The next level above cells is the tissue (Figure 3).

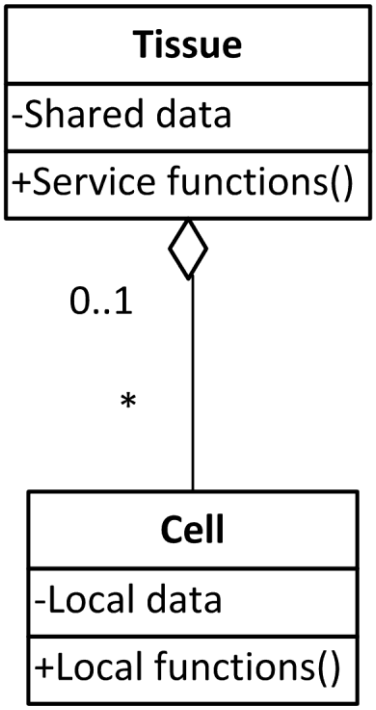


Figure 3 – Hierarchical structure of cells and tissues in UML.

For a description of the notation, see appendix A.

Every cell inside a multi-cellular organism is inside a tissue. This connection is realized for both directions, every cell has a pointer to its "parent" tissue, and every tissue has a chain of links that point to its cells.

A tissue consists of cells that work together closely. In traditional programming sense, this can be thought of as a function that calls smaller functions. In this case, there is a main cell that more or less corresponds to the bigger function, and this cell uses the "service" of the smaller cells, and those might use other even smaller ones to fulfil their tasks.

Different cell types are stored in associative containers that can be addressed with the unique type ID of the cell type. This ID is generated automatically with

every run of the VLO. With this value, the set of cells of that particular type can be accessed along with a few shared values. This data can only be used by the tissue functions. The cells cannot access it to prevent unnatural cheating by cells (like moving all the cells at once to other tissue or killing them). The type ID can be looked up from another associative container that returns the ID number from the name of the cell type. To prevent confusion of different naming conventions of cells created by different programmers, the cell name is generated from the cell class by means provided by the programming language. The cell set is used by many functions for example to return the cell count of a certain type or during hibernation. Jobs for cell types are also stored in associative containers whose keys are the cell type IDs (Figure 4). These containers return a queue of all the unprocessed jobs of the cell type. The tissue provides functions for cells to handle these jobs. There is an inserting function that inserts the given job to the cell queue of the given cell type ID, and job fetching function that returns the first (longest present – first in - first out type) job of the given cell type ID. The tissue can also report the number of jobs waiting inside a cell type's queue. This is used in the maintenance process described in the previous chapter. Jobs sometimes need to go to other tissues. In these cases the address of the recipient cell needs to be given in the from of cell type ID, tissue type ID and organ type ID. These IDs can be looked up with the helper functions of the tissue form the type names in the same manner as described above with cell types. Several associative containers exist in each tissue for this purpose. To help blood cells go to the next tissue in the circulatory system, the tissue can point to the next tissue with a function. There is also an explicit function that helps to transfer cells to the given tissue. For further details, see the Chapter 4.

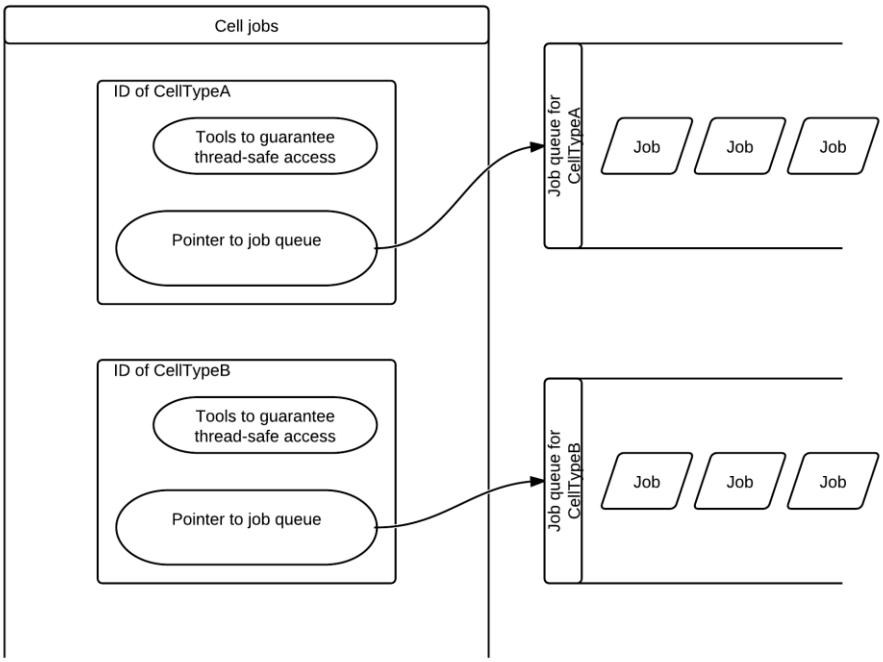


Figure 4 – Structure of the container of cell jobs inside a tissue

Tissues also play important roles in creation and hibernation. The tissue can add new cells and cell types to itself; it can start threads of a given type of cell, join (end) threads of cell types and help cells to die. When a cell dies, it is not preferable for all the cells of a certain type to die, as in that case no one would check if a new job has arrived for that type. For this reason, a cell can ask the tissue if it can die and if all the conditions are met, the tissue helps it to die, otherwise it denies the request.

3.3 Organ

Again starting with the definition of organ as found in the Merriam-Webster dictionary:

“a differentiated structure (as a heart or kidney) consisting of cells and tissues and performing some specific function in an organism”.

The main property is that an organ consists of tissues and cells, and it has a main purpose, a specific function. From this point on, the word organ will mean the virtual organ, unless it is specifically expressed that the biological organ is meant.

In the VLO, tissues are stored in organs; every organ has a main tissue that realizes the main function of that organ, and several support tissues that can support maintenance and any needed and pre-defined service that might be required by any cell in the organ to function normally. Every tissue has a pointer to its organ, and every organ has a pointer to its only main tissue and a set of pointers to any support tissues it might have. The support tissues are stored in an associative container with their unique IDs as a key.

Organs are indeed needed structural elements. When cells work closely together, a tissue is enough to facilitate this. But there might be times when cells access indirect services, like a globally synchronized variable, that actively needs to be synchronized, this service then needs to be done by a cell. The synchronizing cell is strategically different from the other working cells, so it's not logical to put it in the same tissue. This is where the support tissues come in. They have all the maintenance and support services that make a cell's work possible. On a strategic/functional level, an organ can be thought of as not much more than its main tissue extended with only the necessary, most of the time rather invisible services. (One example for this globally synchronised variable is determining the number of current cells inside the organism. This subsystem uses organs to store its values and has a separate support tissue that updates this value periodically.)

Organs, similarly to tissues have a pointer to the next organ inside the circulatory system. The difference is that this link belongs to the outer circle of the circulatory system. Further details can be found in Chapter 4.

3.4 Organism

As with all the concepts and entities, multiple sources can have different definitions of them, most of the time just with different emphases of different parts or properties, but sometimes even including or omitting certain properties. The quoted definitions before included all the necessary parts that were found in definitions in other sources, that is why a single source was selected as the Merriam-Webster dictionary, but the word organism has two important definitions even in that source, with important details in both of them. So organism is defined as:

“a complex structure of interdependent and subordinate elements whose relations and properties are largely determined by their function in the whole”

and

“an individual constituted to carry on the activities of life by means of organs separate in function but mutually dependent”.

The first definition highlights the hierarchical structural makeup of the organism, and defines the relations and properties of the elements, namely that their function is what defines them. The second definition makes it a bit clearer that the main components are organs that should be mutually dependent on each other. The most important detail is that the organism should do the activities of life, so it is a living entity and should have the properties outlined in the subchapter about cells. From this point on, the word organism will mean the virtual organism, unless it is specifically expressed that the biological organism is meant.

In the VLO framework the topmost structural level is the organism. An organism consists of organs. Multiple organs can create a logical set, the organ system. For example, an organ system might be a user interface or database handling. An organism is a process from the operating system's point of view.

The organism holds organs much in the same way as organs hold support tissues, in an associative container addressed by unique IDs. Presently, organisms do not have any other functions, although as it can be seen by past examples (e.g. determining cell numbers), this could be the place for truly shared variables that needs to be stored in a central place. The hierarchical structure of the whole organism can be seen in Figure 5.

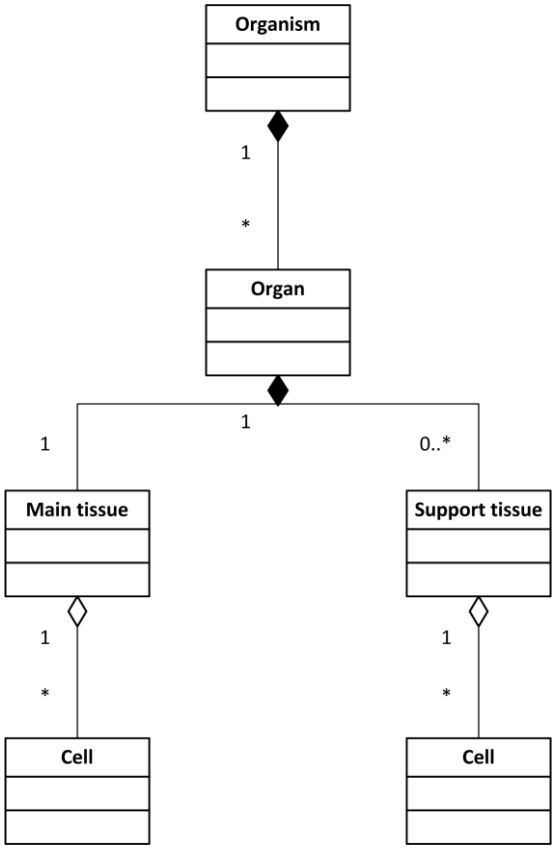


Figure 5 – Hierarchical structure of a virtual living organism in UML.

4 Communications inside the organism

4.1 Communication types

In nature, there are many types of communication between cells inside an organism. The vast majority of this communication is done via chemical signalling. The signalling mechanism inside the virtual organism has the same types of signals that we see in biology. The biological types and their properties can be seen in Table 1.

Communication type	Target and mode of transportation [a]
Endocrine	Any type of cell in distant target tissue, reached using the circulatory system
Paracrine	Neighbouring cell of any type within tissue or organ
Autocrine	Same type of cell, transported outside of cell
Intracrine	Same cell, transported within the cell

Table 1 – Intercellular communication types and properties in biology

[a] from (Nussey & Whitehead, 2001)

The virtual chemical used for communication is the job data structure. As intracrine signals do not leave the cell where they were created, and are used for conveying information between the parts inside the cell, and the smallest part simulated widely inside this virtual organism as part of the framework is the cell, we do not need to use this type of signalling we can think of intracrine signalling as being represented by the use of variables inside the program of the cell (as long as a more detailed method is not needed).

An autocrine signal acts on the same cell that created it, so this is realized by sending a job for the same type of cell that created it, and sending it to its own tissue (Figure 6). This is widely used as part of the death (apoptosis) process, where a hidden signal (i.e. access to this information is restricted) is embedded into the job structure, and when a cell processes this job from the queue, it re-emits the signal so other cells of the same type can process it too.

Paracrine signalling is used to reach any type of nearby cells; this is implemented in the virtual organism as a direct way a cell can use to reach other cells inside the same tissue (Figure 6). In this case the goal of the simulation method was to use the inter-cellular substance inside the tissue to convey the information. This is done by the cell inserting a job for any kind of cell inside the tissue by using the job inserting service of the tissue directly. This function inserts the new job to the target cell's job queue securely. This should be the most widely used method of signalling inside a typical organism, and is optimized to be very fast compared to more complex communication types like the endocrine signalling.

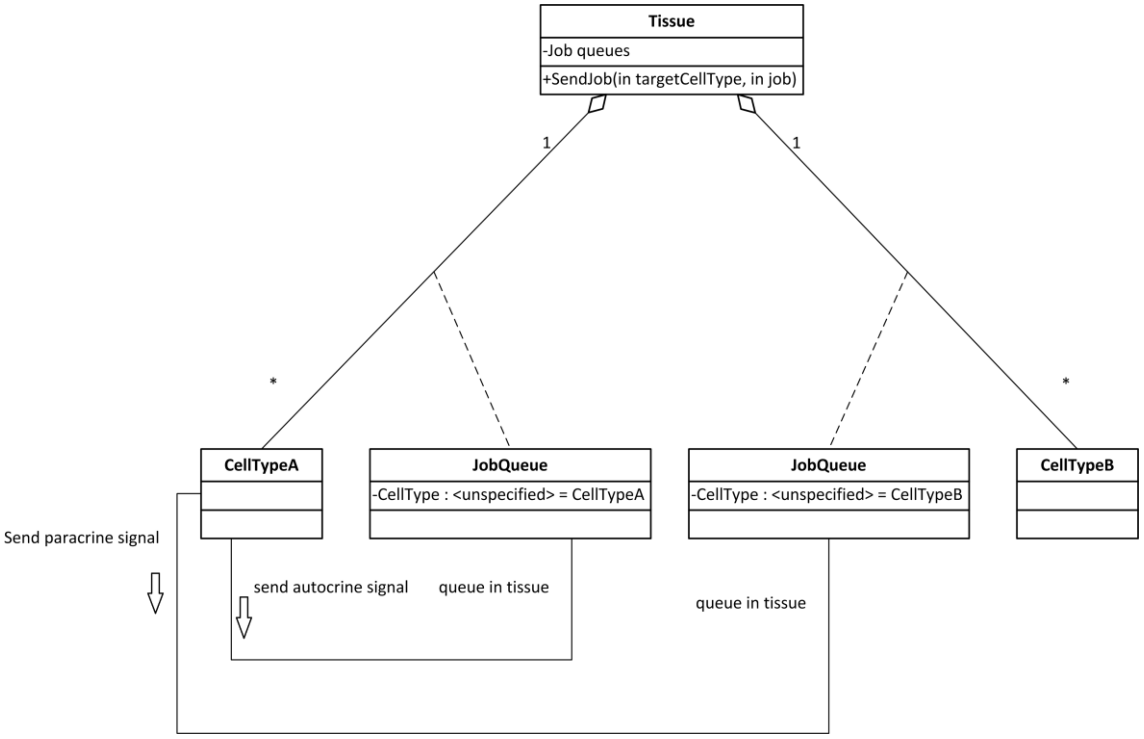


Figure 6 – Paracrine and autocrine signalling inside the Virtual living organism

4.2 Endocrine signalling and the circulatory system

When the target cell type is outside the tissue of the message sender, the endocrine signalling system is used. In real organisms this makes use of the vast circulatory system inside the organism that can reach all of the tissues. In our VLO this system has two layers, the first connects all the organs, and it is implemented as a pointer in every organ that points to the next one. The second layer goes a circle inside the organ (Figure 7) that starts with the main tissue of the organ, as it is expected that the majority of the communication done inside the organism will have to do with a cell inside the main tissue of an organ. Some of the cells can make shortcuts and skip the inner layer only making a stop at the main tissue of the organ and if it is not the destination organ then they can skip its tissues. Inside every tissue, there is a similar pointer, as in the organs, that points to the next tissue of the second layer.

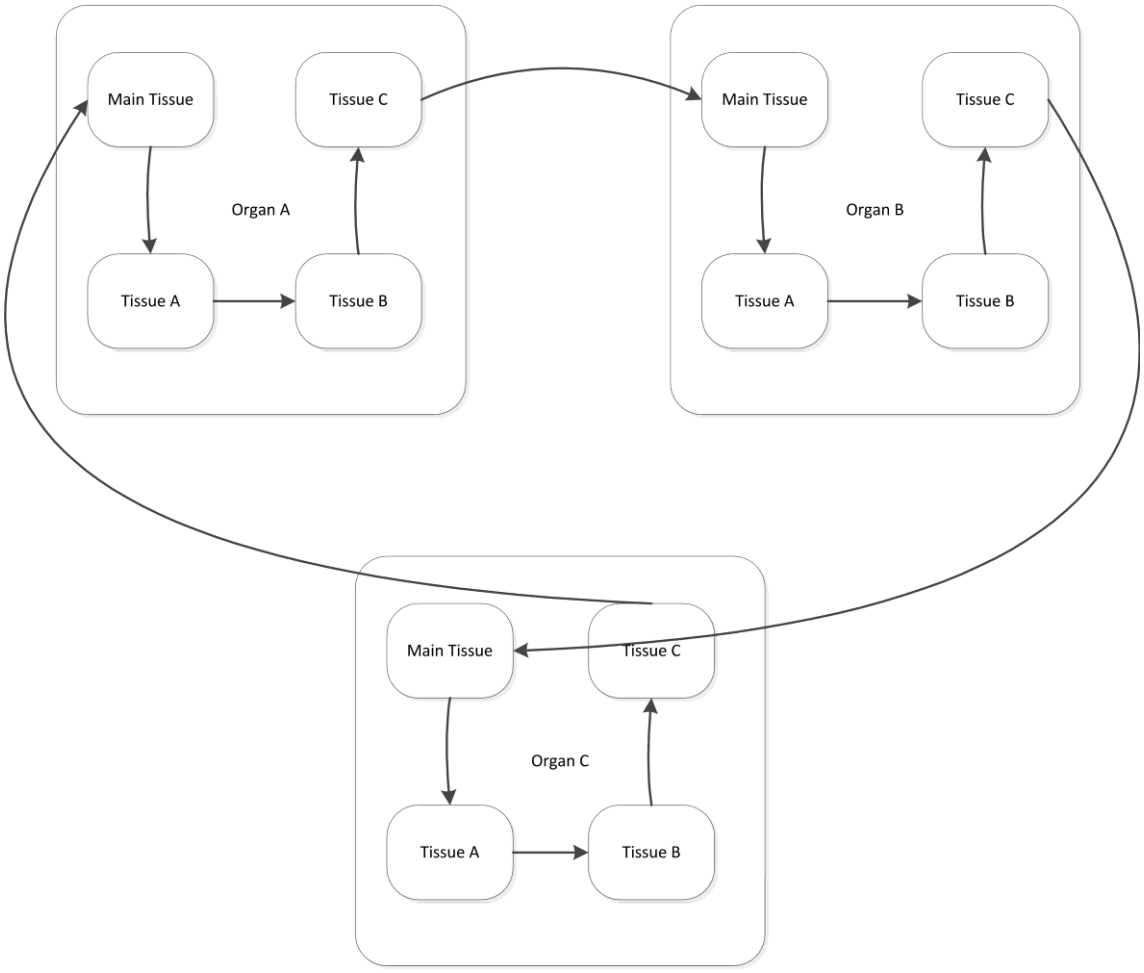


Figure 7 – Schematic overview of the two-level circulatory system inside a Virtual living organism

The circulatory system is almost exclusively used by the blood cells. A blood cell can be given a job with details of the target and the job structure as the message. The address must include the target cell type, and can include the organ and/or the tissue name that has the cell inside it. If there is a specific address (i.e. not just a cell type, but an organ or tissue it resides in), then the blood cell can skip the organ if it is not the target by not going through its inner link, and can detect if it went through the address and the target was not there (i.e. the target tissue did not recognize the target cell type). When the blood cell finds the target cell type either inside the specific tissue or organ when given, or otherwise after going around the circulatory system and searching for the target

cell type (this is done by "asking" the tissue if it has the type of cell), it inserts the message as a job to the target and continues its work.

In nature, most of the chemical signals travel as dissolved molecules inside the blood stream. In this emulation, there is no physics simulation embedded, so no passive parts - like those dissolved molecules - can move by themselves, only active parts like blood cells can interact with them. As long as this inaccuracy does not cause a problem in a given emulation task, this system is rather similar to what actually happens in nature. As messages appear inside the tissue fluid, they indeed have similar properties to dissolved molecules. These molecules get into the blood stream automatically by a function in the tissue called by a cell. As they travel around the circulatory system, they get inside tissues with a selection, like ultrafiltration in nature, and bond with the target cell by being put into its work queue. It is easy to see the similarity of this model to the natural circulatory system if we imagine a copy of the **BlooDCell** type and name it DissolvedMolecule, it could use the same routines as the blood cells do, and use the circulatory system the same way. This would be closer to nature, but as long as this naming is not a problem, having the **BlooDCell** do the transportation of all molecules may make it clearer how the system works.

There are two types of blood cells at the moment, one is called **BlooDCell** and is used as described above, the other is called **SpecialBlooDCell**, and this one is used to distribute information to all cells of the same type inside the organism, mimicking a similar process in nature (Valitutti, Müller, Cella, Padovan, & Lanzavecchia, 1995), although in nature this feature is realised with a chemical and not a cell. This is done by a **SpecialBlooDCell** going around the circulatory system, adding the job to every job queue that corresponds to the target cell type and going on until it reaches the starting point, when it gives a job to the originating cell notifying it of the completion of the job. This process can be used for synchronizing certain events or values inside the whole organism. For example, a signal that is given to cells to notify them at the beginning that it is safe to start because a consistent environment has been set up. The **SpecialBlooDCell** has an option to wait at every step for a new job to continue

its journey with. This allows for gathering data from all the cells of the same type. It is intended to be used with an initialised empty set or similar type that is being updated at every step. This process is used, for example, to gather all the type names of cells present in the organism after birth.

4.3 Jobs and results

Job structures are completely opaque (no information is known of them outside of the cell type they belong to), and can be created by calling the **jobOrder** method of the target cell type with all the data that a cell needs for doing the required job. This method returns the opaque structure that can be sent using any of the communication methods described in this section. Job structures are, as their name suggests data structures. Each cell type can define their own structure, but must base it on the default Cell type's job structure as it contains default flags for shutdown requests and kill requests. This base structure is then extended with all the job information the cell type may require, and the cell type has to provide a **jobOrder** function that copies this data into their correct place in the structure and return the newly created structure.

There is a method of returning results of a job that is implemented here, but it is implemented for the sake of simplicity and not necessarily as an accurate representation of a real biological process. Inside the job order, it is possible to insert a special type of "pointer" that contains the address where the results should be sent to directly. The sender can then wait until the results are back and continue its work using them. This mechanism is not a problem in terms of biological accuracy, as it can be chosen not to be used if unneeded and the same result can be achieved if the sender cell is made into two separate type of cells, one doing the work only before sending the job, and the other doing the work only after receiving the results. This latter aproach is biologically more accurate and achieves the same result, but is harder to maintain and handle, so as a simplification method a special pointer can be used when it does not pose a problem. The results can be received in another opaque structure that can be opened with the **extractResults** function of the result sending cell type.

4.4 Communication example

Sending information to a cell inside a different part of the organism is a complex task. Now that all the parts have been detailed, the only question left is how does this process come together? How is a single communication happening actually? Here is an example of **CellTypeA** in **OrganTypeA**'s main tissue sending a job to **CellTypeB** in **OrganTypeB**'s support tissue named **STissueTypeB**, and getting back the results.

The communication part starts at **CellTypeA**'s main function, where inside the main loop; the cell decides to send a job request to **CellTypeB** with the values 3 and "apple". The cell of course knows what **CellTypeB** does and requires a result from it. For example this scenario can happen if the values represent a database entry and the results tell if it was successful. The first thing to do is to create a job structure with a call to the **jobOrder** function of **CellTypeB** with the input and a pointer for the result (this part is partly pseudo code for the sake of readability). This would look like:

job=**CellTypeB**::**jobOrder**(3,"apple",resultpointer). Then all the cell needs to do is send this information with **CellTypeB** written in the address field: **addJob**<**CellTypeB**>(job). This convenience function calls the local tissue's **addJob** function that first tries to locate the recipient inside itself, and if fails it gives a job to the **BlooDCell** with **CellTypeB** as the address and job as the job. The **BlooDCell** first checks if the address field is properly filled in as be it as possible to give more precise addresses like concrete tissues or organs, and if it finds everything satisfactory, it starts searching for the given cell type. It starts its search in the current organ's main tissue as a part of a performance optimisation. It transfers itself to this tissue with the help of the tissue's support functions, and asks the new tissue if it has any of the target cell type. If not, then it transfers itself to the next tissue in the inner circle of the circulatory system. This is repeated until it arrives back at the main tissue of the organ, when it no longer checks for the target, but asks for a transfer to the main tissue of the next organ in the outer circle of the circulatory system. This process is repeated until

the recipient is found. If, as in this case, there are no more specifics in the address then this only means the first instance of the recipient's cell class is found. After finding the right tissue, the message is inserted by the blood cell into the job queue of the recipient cell type. The next part of the process is carried out by an instance of **CellTypeB**. During its main cell loop, it waits for an incoming job. When the new job arrives, the cell is woken from its sleep. The first thing it does is to ask for the new job. As the cell knows its own job structure, it checks the shutdown and kill flags and if none of these are set, it gets the data out of the structure and starts working with it. When everything is processed, it fills the results into the return structure and sends a direct signal to the sender of the job that signals if the result is ready. This of course is again a way that makes the system faster and can only be used if this biological inaccuracy is not a problem. If it is, then the result can be sent back as another job. When **CellTypeA** receives the signal, it is woken and can use the **CellTypeB**::**extractResults**(resutlpointer,data) call to get the results into the data variable. At this point, the sample communication session is concluded.

As can be seen by the example, the endocrine signalling process in the VLO is very complex, as in nature, but in reality it still remains fast enough to be used as the main type of communication between distant cells.

5 Life cycle

Life cycle in the VLO means two things: the creation and hibernation of the organism itself and, as part of that, the creation and death of the individual cells. There are some temporal and quantitative differences between the two as the organism only is created and hibernates once in its lifetime, the cells within it can be born and can die many times due to the self-replication and maintenance process. There is one cell type, that has special significance in both processes of the life cycle, and this cell is called **DivineCell**.

5.1 DivineCell

As previously mentioned all the physical work in the VLO is done on a cellular level. The creation and hibernation processes are complex enough to require at least one dedicated cell for this purpose. The **DivineCell** (**DC**) is a cell type that can be thought of as a version of the stem cell in biology. In fact in its purpose of helping in creation and hibernation (somewhat similar to death in its role) it is more like embryonic cells in biology. The reason that it's not called StemCell is that it is heavily optimised for the virtual environment, and also has a slightly different role, and such a name would cause confusion. In the future there might be a virtual version of the real stem cells, especially since the subject is heavily researched and the VLO framework by its nature would be a great emulation tool for this. The name **DivineCell** comes from the fact that a VLO unlike its biological counterparts has no parents. The advantage of having a parent comes from the evolutionary process, and as the use of the VLOs is mainly a onetime emulation, they need not to evolve. Instead they use what can be called the shorter version of evolution: adaptation (Sommerhoff, 1950).

Since there are no parents from which the VLO can come from, the organism has first to be put into existence by a "divine" act, hence the name **DivineCell**. The **DC**'s task and "scope of authority" fits the hierarchical nature of the VLO well. There are many **DC**s inside an organism, and each is responsible for the hierarchical subtree it is the root of (i.e. the tissues and cells it is responsible

for). There is one **DC** that is different from the others; this is the main one responsible for the whole organism (Figure 8). The others are responsible for one tissue each.

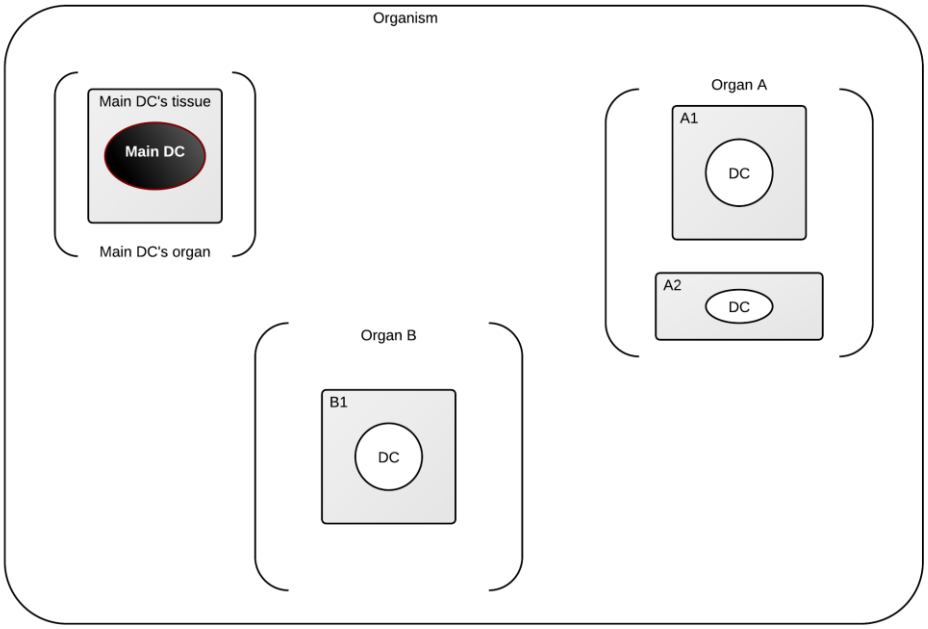


Figure 8 – The virtual living organism and its DivineCells

The organism (rounded rectangle) has many organs (brackets), the Main DivineCell creates all the organs, and with them the tissues (shaded boxes). After creation, each local DivineCell (white circles) is responsible for maintaining their tissues (light shaded rectangles).

The job of a **DC** is to help both the creation and hibernation processes and act as a registry for cells. This is needed because of two reasons. First, as opposed to nature, the virtual organisms' hibernation process must end in a consistent state, meaning that there should not be any data loss and resources must be freed. This means that when the user or the main task of the organism signals

that a cell is required by to shut down (hibernate), it is not acceptable to kill all the cells immediately. They must be informed of the shutdown request and the system must wait for them to reach a consistent state (see the "hibernation" section for details). This requires a registry of all cells to be able to wait for all of them before the main process is ended. The second reason why a registry is needed is that when a cell dies because it is not needed anymore (see details in the self-replication section), it cannot fully delete itself and free up all the resources it used by itself. It is the **DivineCell**'s responsibility to finish the death process of the cells it is responsible for. In nature this registry function is not needed as there are many processes that have monitoring and active nature. Stem cells monitor damaged cells and help repair them. The approach of active monitoring however wastes resources and is replaced in the VLO by passive solutions instead that work with signalling for example when a cell dies. This can of course be replaced in the future if the active approach would change the outcome of the emulation. In the creation of the organism, the main **DC** has the task creating the basic structure, the organs and the tissues inside them. After this is done, it waits for the **shutdown signal** from one of the cells, it helps to propagate it and waits for everyone to finish their jobs and shut down properly. Then it signals to the operating system that the program has finished its work and can be shut down now. The tissue **DC**s create all the cell types and store the cells' data to help free up resources when a cell dies. If it detects a **shutdown signal** by one of its cells, it sends it to the main **DC** and waits for the propagated **shutdown signal**. Then it sends the signal to all the cell types and monitors the situation. When all of its cells are shut down, it sends a ready signal to the main **DC** informing it that the process can continue. A more detailed description of both events can be found in the next sections.

All **DC**s are subclassed from the same **DivineCell** class and have many convenience functions to enable rapid productions of these mainly by just telling them what kinds of organs/cells they need to create, which makes their creations a rather automatic process if there are no special circumstances. **DC**s only communicate with other cells as all the other cells in general, by sending jobs. Their job structure is standardised and needs no extra programming.

5.2 Creation

Creation here means how the virtual organism first came to exist. It is the process that starts with the user starting a program and ends with a virtual organism in existence on the same computer. This process is partly like birth in biology but, unlike it is with complex biological beings who are first in an immature state for a rather significant proportion of their lives, the virtual organism has to become fully operational in as short a time as possible. This means that, unless it is a requirement for a simulation to have an accurate birth and maturing process, the VLO benefits from a creation process optimized for speed.

For building the structure of the organism, there are two possibilities, a top-down or a bottom-up approach (Figure 9). The former would mean that first we create the organism, then all the organs and tissues, and after this, the cells within them. Most of the time this is the approach used in programming, since it means that programmers can create self-containing structures, for example an organism that automatically creates the tissues it needs, and the tissues can create the cells they need. While this would be quite logical, and since everything above the cells is almost purely a logical level, the biological equivalent would be more or less physically plausible, but it is not how this is done in nature, and it would be hard to extend to a more accurate emulation. So our implemented approach is a bottom-up approach, or rather a hybrid approach that starts with a bottom to top part and still makes it possible to create self-containing structures.

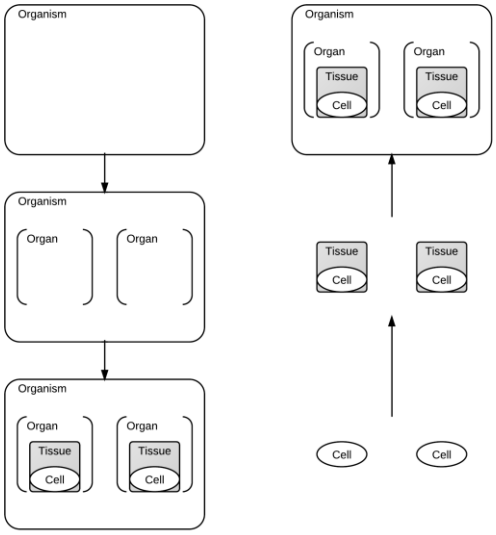


Figure 9 – Virtual living organism building approaches.

Top-down approach (left), bottom-up approach (right).

The first step must be to “magically” put one cell into existence, and this must be a special kind of cell whose main purpose is to create a virtual biological structure. This cell is called a **DivineCell**. When it is created by the program, it does not have any structure around it, so at this stage we speak of a single cell organism. The first thing this cell has to do is to create its own immediate environment (see Table 2 for the entire process). It creates a tissue, an organ, an organism and sets up the correct relations, properties of these (unique IDs, relations and the link of the circulatory system). Then it is time to create the other custom organs. Now these can be self-containing, meaning that when a specific organ is created, it automatically creates and sets up all the tissues inside it. And as the tissues are created, a specialized **DivineCell** is automatically created inside them that will be responsible for setting up and maintaining its own part of the organism (meaning a tissue and whatever is inside it). At this point, the process becomes multithreaded, this first part describes the main **DC**’s involvement in the creation, then the tissue **DC**’s tasks

will be detailed. However if these independent **DivineCell**s would immediately start creating all kinds of worker cells, they would try to start using services of other parts that might not exist at this point, for example because those cells that these services rely upon do not exist yet. This brings about the need of a synchronization process after creating the cells, but before making them active. This synchronization process has several steps and relies heavily on the **SpecialBlooDCell** introduced in the Communications section to gather and distribute data, and to send out signals for time synchronization.

	Process	Actor
1	Creation of main **DivineCell**	Main program initiated by the user
2	Creation and setup of main **DC**'s organism, organ and main tissue	Main **DivineCell**
3	Creation of custom organs	Main **DivineCell**
4	Creation of tissues and local **DivineCell**s	Custom organs
5	Creation and setup of local cells	Local **DivineCell**s
6	Initial synchronisation	All **DivineCell**s in the organism
7	Start worker cells	Local **DivineCell**s

Table 2 – Main processes of creation

The first part of the synchronisation sets up the Logger cell (see Chapter 6.1) so the process can be logged, and a **SpecialBlooDCell** used throughout the process. Then it sends the **SpecialBlooDCell** around with an empty data storage that gets filled in with all the tissue and cell types known by all the **DC**s in the organism (this means all the types they are responsible for). When the result arrives back, it stores this information, sets up the global cell-, tissue- and

organ type IDs and sends the updated data around so all the **DC**s can put this information inside the tissues for later use in messaging. With the first results, all **DC**s also have sent back their thread handles, so the main **DC** stores this as well to be able to detect when every other cell has finished their work during the shutdown process. When a tissue **DC** has received the global IDs, in theory it should be able to start its cells and they should be able to start working. The only problem is with cell migration. Currently only the **BlooDCells** migrate to other tissues, but even now if a message is sent the blood cell would arrive at a tissue that still does not know what a cell type ID corresponds to and it would cause an error. So it is crucial for every **DC** to wait until all the others have received the update. For this purpose, the main **DC** waits until the **SpecialBlooDCell** with this update arrives back, and send around another round with an “all ready” signal and the tissue **DC**s can finally start their cells. After this, the main **DC** enters its main cycle waiting for the shutdown signal. This part of its operation is described in Chapter 5.3.

Tissue **DC**s start by setting up their own local environment, but they already have a global environment unlike the main **DC**, so they only have to set up properties and relations of their tissue and themselves. After this is done, they wait for the first part of the synchronisation and insert the cell names they know plus their own thread handle. Then, while they wait they create the blood cells for their tissue, and when the global IDs arrive they set them up in the tissue. Then while they wait for the “ready to go” signal, they create all the cell types that should be in the tissue and when the signal arrives they start all the cells. Then they enter their default run loop that will be described in Chapter 5.3.

The only part left to be described in the creation process is cell creation by the **DC**s. When a cell is created, the required information is the cell type and the number of cells to be created. First the **DC** checks if the tissue already has this type of cell. If not, some data needs to be set up, like the job queue for this cell type. If the cell type is known, then it creates the new cells, sets up their properties, inserts them into the tissue and reports the cell number change to all the places necessary (see cell number synchronisation in Chapter 6.2). When

all the new cells are created, it calls a function of the tissue that starts the threads of all the cells of a specified type that has not been started yet. With this, the new cells are created and running.

5.3 Hibernation

Hibarnation in a virtual organism has a similar meaning to death in biology. That is because of our expectations for it. In nature death can be thought of as an event when the living system comes out of its dynamical equilibrium, its consistent state of being alive. It is easy to simulate the event of death, we only need to cut off the virtual system from its resources, e.g., terminate the main process containing all the threads, or just terminate one or more threads and do not worry about any consequences. The only problem is that with a computer program, we demand it always to be in a consistent state, especially before its termination. For example this might mean saving data to a persistent location, writing out cached information to disk (think of logging data in the form of jobs waiting in the job queue of the blood cell in a tissue waiting to be delivered to a logger cell somewhere else), or just simply to wait for processing the last of the data when a data factory cell signalled the death signal after it finished creating all the data to process. This form of termination in its details works more like a hibernation process in nature, when preparations need to be done for switching to an alternate state of unproductive consistency. At the same time it differs from hibernation in the way that hibernation has a semi-persistent internal state that remains unchanged after coming out of hibernation, while a virtual organism is being reborn after this expiration and even if it uses a set of persistently stored information, it is essentially a new specimen with no part of the custom inner state of the expired specimen, although a data storage for inter-specimen cell state saving is planned for the future.

As a result of the consistency requirement, the hibernation process must have multiple stages. This is a somewhat similar approach to that used when operating systems shut down. The first step is the signalling of the shutdown

request that originates from a cell. This cell can emit this signal either as a result of a user request, or because of the logic programmed into it. The request must be sent to a centralized location, namely to the main **DivineCell** via the local **DivineCell** inside the cell's tissue. The tissue's **DC** at this time is in its main loop, as it was at the end of the creation stage. This loop only exits if it detects a **shutdown signal**. Before exiting, it sends the signal along to the main **DC** and proceeds with the shutdown process. The main **DC** receives the signal while it is in its main loop doing nothing but waiting for this signal. When it receives the signal, it sends it around the system with a **SpecialBlooDCell** that does not wait for any answer, just drops it in every **DC**'s job queue inside the organism, similarly to an existing process inside biological organisms (Valitutti, Müller, Cella, Padovan, & Lanzavecchia, 1995). Then all these **DC**s exit their main loops and, not knowing where the job came from, they all send it to the main **DC**, but these jobs will be discarded when they are received. This method might not be optimal, but the number of tissues inside the organism is rather small compared to the number of cells for example, so the extra communication is of negligible amount. After this step, a rather complex synchronisation subprocess starts that's only aim is to allow all the cells to finish their work. One of the main challenges was to allow all the blood cells to transfer the remaining jobs inside job queues. Since blood cells are not tracked, it is hard to determine where they are and if they are working. Also a job waiting to be sent might come from a cell that must receive a reply before being able to shut down properly, so it is crucial to give enough time for these cells (Figure 10).

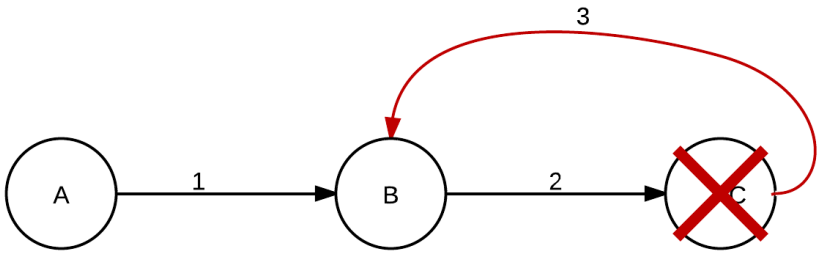


Figure 10 – Effects of early cell death

If cell (circle) A dies, then every other cell can die any time. If cell B dies, then any future jobs from cell (represented by arrows) A will not be processed and cell C can die any time, any unprocessed responses from cell C will not be processed in the future. If cell C dies before cell B could send its last job, then B will wait forever for the response (arrow number 3), so it will not be able to die until the whole organism dies.

The tissue **DC**s, after sending the shutdown signal to the main **DC**, also send the signal to all their cell types. When the cells look for work and find this shutdown request, they reinsert this request to the end of their queue and try to die. If they would just simply die, it might happen that another cell requires some response from them before they can die, so one cell has to remain alive from all the cell types till the end to process such requests. In this stage, cells are not permitted to create new tasks, just finish processing the existing ones. After the tissue **DC** has sent the signals to the cells, it waits for the main **DC**'s synchronisation. The main **DC** sends around requests questioning if every tissue **DC** is ready to be shut down. When the tissue **DC**s receive this question, they look around and see if there are jobs left in the cells' job queues, and if so,

they are not ready. If there are no jobs, they check if there was any blood cell activity since the last questions, and if so, they are also not ready. If any tissue is not ready, the main **DC** waits for a while and sends the question around again, until all the tissues answered ready, or the predefined shutdown time allowance is spent. In either case, the main **DC** sends around the kill request. This is also propagated to all the cells, and the cells must die regardless of what they are doing. After that, the tissue **DC**s die themselves and the main **DC** detecting this (giving another few seconds to each **DC** to do this) dies and gives the operating system the signal to finish the whole program (See Table 3 for a list of processes involved in hibernation).

	Process	Actor
1	Send shutdown request to local **DivineCell**	Worker cell in a tissue
2	Send shutdown request to main **DC**	Local **DC**
3	Send shutdown signal to all tissue **DC**s	Main **DC**
4	Send shutdown signal to all cells	Local **DC**s
5	Die if not the last cell of its type in the tissue	Worker cells
6	Check if **DC**s are ready to die, if not then wait and try again	Main **DC**
7	Send kill signal to local **DC**s	Main **DC**
8	Send kill signal to worker cells	Local **DC**s
9	Wait for cells to die and die	Local **DC**s
10	Wait for local **DC**s to die and die	Main **DC**

Table 3 – Processes involved in the hibernation of a VLO

This covers the hibernation of the virtual organism, but the death of individual cells is an entirely different thing. As it was mentioned before, cells try to optimise their numbers to cope with changes in workload. The optimisation process itself was detailed in Chapter 3.1 about cells. When a cell type in a tissue has more cells than required for the current workload, the extra cells die to save resources. This process involves the cell itself and the **DC** of the tissue it resides in.

As stated before each cell can “request” death in two ways. One is where it first looks around and only dies if it’s not the only one left. This is to ensure that no essential cell type can die out before the hibernation of the organism. This kind of death is used both in the death caused by optimisation and in the hibernation caused by the shutdown signal. This is only needed if the cell in question is not an equivalent of a singleton type, that is, the number of cells of this type is dynamic. The other kind of death is unconditional and is used at the end of a cell’s life, typically when the kill order is received. Although these are two different death helper functions, they only differ in the check of being the only one, so the extended one will be detailed.

Both death functions are implemented inside the tissue as custom built cells need not to know (and especially alter) how this process works. However a customised tissue can change this behaviour. The function first checks whether the cell is not dead yet which is indicated by a flag associated with the cell. These kinds of extra sanity checks can be found throughout the framework to ensure error-free operation. Then it checks if the cell can be found in the tissue’s registry, if not it returns with an error. If so it checks if there are more than two cells of the type that is about to die and if so it continues on. It then decreases the count of the cell type that indicates how many cells of it are active, and if it reaches zero it sends out a notification that will be processed by the local **DC** when it waits for all the cells to die after the **kill signal**. It then notifies the subsystem that’s responsible for maintaining an overall count of the cells inside the organism that the number of cells here has decreased and the subsystem should report this change. More information of this subsystem can

be found in Chapter 6. After this the local **DC** is informed that this cell has died and the dead flag is set in the cell so if it is examined by another cell they know it died, and the function returns control to the cell. The cell should end its main function at this point.

During this process, the local **DC** should be in its main cycle waiting for either jobs about dead cells or a shutdown signal. When it is woken by the job the cell sent it through the tissue's death function, it tries to remove the cell from the tissue. If this had been done before, the proper pointers would have been severed and the tissue would have some trouble reaching the cell, so this function can only be safely accomplished by another cell and is best done with the cell that maintains the registry, the local **DC**. If the removal process ends with an error, it logs this; otherwise it "joins" the cell which means that it waits for the cell's thread to end. This is required to free the remaining memory and other resources the cell held. After the cell terminated it is deleted from memory and the **DC** waits for the next job to arrive (See Table 4 for a list of processes involved in cell death).

	Process	Actor
1	Check if cell lives and can die	Cell to die
2	Decrease number of active cells of this type	Cell to die
3	Notify cell number synchronisation subsystem of change	Cell to die
4	Inform local **DC** of intention of death	Cell to die
5	Stop working (end program)	Cell to die
6	Sever links between tissue and cell	Local **DC**
7	Wait for the end of cell's work and terminate its thread, free its resources	Local **DC**

Table 4 – Processes involved in the death of a cell inside a VLO

This concludes the process of a cell's death. As it can be seen, the organism's death resembles a well synchronised process like hibernation, while a single cell's death during the lifetime of the organism starts a process that's main reason is to disassemble the dead cell's body which can be credited to the fact that there's no physics simulation in the background that would normally decompose the dead cell to small enough parts that can be carried away by the intercellular substance. In a way since the tissue is responsible for the handling of this substance it is well justified to have a function that does this job and by doing so it plays its part in the physics emulation (as opposed to simulation) that keeps the system going.

6 Subsystems

This chapter details some components of a VLO that can be called subsystems. The biological equivalent would be the organ system (or biological system), although these subsystems are not necessarily built up of organs only. Sometimes every organ or tissue that wants to use the subsystem's services must have a tissue or cell inside them that communicates with the base of the subsystem in question. These subsystems are not necessary for an organism to function properly, but can add new functionality or can have qualitative benefits like speed optimisation. As such they can be viewed as not part of the base of the framework that describes and implements the programming paradigm, but as bundled examples and optimisation routines. The only exception to this at the moment is the cell number synchronisation subsystem as it is integrated into the framework to a greater extent to prevent the VLO from using more resources than what is available in the host computer, but this problem can be handled in many other ways as well if this is left out.

6.1 Logging subsystem

The logging subsystem is considered to be the main way of communication between the VLO and the user of the computer in the current examples. This subsystem logs many kinds of messages sent by cells of the organism. These messages can be classified into two groups. One consists of automatic messages that can be used mainly for debugging, these messages are generated by the superclasses (parent classes) of cells and contain information like the creation or death of a cell and may contain other information that might help to identify the cell, its type or the conditions of the action. The other group consists of custom built messages that every cell can generate to report results, data or anything the programmer needs as an output.

All log messages consist of four parts:

- **Message ID**: this identifies what the message holds and can be used for automatic processing of the results. It can have the following values: **creation of**

a cell, **death of a cell**, **sending a job**, **receiving a job** and **generic** that indicates a custom message.

- **Cell ID**: the sender of the message, consistent within the lifetime of the organism but changes between organisms.
- **Timestamp**: the time when the message was sent. The precision of this changes from platform to platform.
- **Message**: the text of the message.

There are two types of loggers, both are just one cell that can reside in any tissue/organ, but for the sake of being optimal, they are created by the main **DC** in its own tissue as normally this place would have no communication traffic before the shutdown, so it would not create any traffic jam. The older and simpler logger is called Logger and it logs all incoming messages to the standard output that is by default the screen. This logger is easy to set up and does not need any extra resources to do its job but can only be used effectively in very small organisms as even during testing some organisms were generating thousands or even tens of thousands of messages in a matter of minutes. Furthermore this logger could only exist as a singleton because if there are more of these, then the output becomes an intermingled mixture of messages, essentially useless. So it was necessary to create a more sophisticated mode of logging which resulted in the creation of the DBLogger cell.

As the name suggests, the DBLogger writes its logs in a database, namely an SQL database, and as such in theory (after some small adjustments) there could be more of these if required in one organism (this would be more useful in for example a distributed organism across many computers). The DBLogger also logs with every message the current **runID** making it easy to filter just the output of the last organism.

To further optimise the speed of the logging process (that can generate huge amounts of jobs), the blood cell was modified to be able to carry out bulk jobs. It looks at its job queue and pick up the first batch of messages that have the same destination and delivers them all in one go. This behaviour can of course be disabled if it leads to any inaccuracy in a simulation, but it speeds up the simulation significantly as it decreases blood cell movements considerably.

6.2 Cell number synchronisation subsystem

Nature uses parts that can be translated into computer science (CS) terms to little computers, or to be more precise and faithful to the current programming paradigm, little processing cores, a great many of them. On the other hand, computers do not. Home computers tend to have less than 10 processor cores for general purpose computing. The difference is great and this can easily distort the model one wants to build for emulation. Clearly, cells work independently and as such they would naturally have to get one separate line of execution – a thread – each. This may seem exceedingly too much as theoretically a computer cannot do more simultaneous work at one time than the number of processing cores it has. Anything more would mean unnecessary switching between threads that increases the overhead of execution. This overhead is not significant enough to be worth creating a thread for drastically (and functionally) different program parts, and this leads to the situation where every-day user programs tend to have just less than a hundred threads.

But where is that limit that differentiates useful functional separation and resource waste? Usually asynchronous operations are put in different threads in programming, like user interface and data operations that should work independently of each other, but surely cells fit this criterion. Even though the thought of having ten thousand threads seems alarmingly wasteful, the thing to remember here is that this is an emulation, not a simulation. Normally there would not be nearly that many cells because of the abstraction, one cell per cell type is enough for many simulation; this is only a safety precaution to check this limit. In a normal VLO, the number of cells, and thus the number of threads

equals roughly the number of cell types added in each tissue. There might be more than one cell of a type in the beginning for various reasons in one place, but since this emulation is built on a hierarchical model, normal operation does not require many more active cells at one time. In the test organisms this meant the cell number was in the tens, but even more complex organisms can be built of around a hundred cells. When will then this increase? Only with self-replication, and even then it should not increase much. Normally the idea was to increase the number of cells of the type that has jobs piling up continuously to reach a state where their proportion is enough to cope with the workload. This means that first it is only reasonable to multiply to numbers compared to the total number of cells inside the organism and that the only goal is to reach the average processing speed of cells inside the organism. If there is an organism with only ten cells then there is no need to replicate a thousand times, only if the cell in question is about a thousand times slower than the others that give it these jobs. And this scenario only indicates badly designed cells and relations that would not work in nature. The result is that normally there would only be a minimal increase of the total cell size, let us say a two fold increase, and even that only if there is a spike of workload for a particular cell numbers. After that spike is processed, the sudden increase moves back to an equilibrium that has the minimal number of cells that enables every type of cell to work with equal efficiency.

Why is it needed to think about the extremes then? Some of nature's processes try to take advantage of strength in numbers. One example seen in this document is the cancer emulation. Cancer cells self-replicate without any restriction imposed on other cells for their own (and mostly the system's) good. Similar processes can be seen in external infections. This instance reached the maximal number of cells in the system which also acted as a suppressant for the self-replication of other, healthy cell types. Even in this instance, it does not really matter how many cells there actually are if they already surpass the number of other cells almost a thousand times.

Why even this many cells? Why is it needed to have all these cells represented with a truly-asynchronous structure if it will be executed semi-asynchronously by the smaller number of processor cores? The answer lies in the details of cell operation. Most cells are waiting for a job to arrive for a large proportion of their lives. The main **DC** for example works actively only in the beginning and at the end. During the lifetime of the organism, it is waiting, essentially sleeping. From the computer's point of view, this thread is inactive and does not require any computation, it only consumes a small amount of memory in the system, so it can be considered cheap. The system can become even cheaper if a thread-pooling system were implemented, that is inactive threads would be cached out, in essence hibernated when they are not working. This would eliminate the notion of a maximum number of threads, but as stated before, that should not be reached only in exceptional circumstances. It could also be that there is just let us say one thread per tissue or even organ, and all the cells share it's line of execution, but certain technological features can make the current solution much-much faster. There are many solutions in scientific computing to boost the number of real concurrent execution cores for a computation. HPC facilities are used extensively, but are very expensive. On the cheaper side, distributed computing can be achieved even at home, and can boost the speed of execution significantly. The most fitting solution however can be found in many home computers, the graphic processing units (GPUs). These have sometimes many hundred cores that can be used for almost any reason, and (both this and the previously mentioned solutions) require exactly the same kind of already multi-threaded program structure that the VLO has today.

The current structure does have this aforementioned limit in today's computer architectures. The limit comes from both the amount of available memory in the system, although this is less and less relevant these days, but the other factor is the architecture, which allows for a certain amount of memory to be used per program. Each thread of the program shares this block, and thus the smaller amount given to each thread, the greater number of threads can be created. This varies from processor architecture (32 or 64 bits), and of operating system implementation. It is only required to point out a safe value below the maximum

that guarantees that until that limit is exceeded the VLO will not crash because of lack of system resources. After measuring the reachable maximum, the limit was set to 1000 threads for 32 bit architecture systems (also called x86), and 10000 for 64 bit architecture systems (also called x64). As mentioned before, the exact maximum can change within some boundaries between system and system and can be several times the chosen maximum amount.

To monitor whether the organism reaches this maximum, it is required to note the current number of cells inside the system. For this reason, a subsystem was implemented that gathers of the approximate number of cells inside each organ, and sums them it up periodically and distributes this information. To prevent many cells from trying to access this information and having to wait on each other, it was decided that this information cannot be stored on the most central of places and must be distributed. Since it would be best suited to assign a tissue to this functionality, the local access for this information was chosen to be in the organ, every cell that tries to self-replicate has to check this approximate value and can only continue if it has not reached the limit. Of course since this is only synchronised periodically, it is really just an approximation, but since the limit is much lower than the actual architectural limit, this should not be a problem.

As a result, the synchronisation subsystem contains of an organ with a main tissue and a cell inside it as a central hub of synchronisation, and in every other organ a side tissue with two cells that are responsible for detecting changes, sending them to the synchronisation hub and receiving updates from the centre and updating the local value visible for the cells inside the organ.

The organ of the subsystem is called **CellNumSyncOrgan**. This organ consists of the main tissue **MTissueMainCellNumDC** that is the actual hub of synchronisation. It also contains the side tissue **STissueCellNumDC** as all organs should that want to possess the total cell count. Strictly speaking this organ does not need the side tissue as it normally does not have any cells that self-replicate, but in case any of the included cells change to have a self-replication capability, including this side tissue makes sure that there won’t be

any problems. There might also be guest cells that self-replicate, like cancer cells.

The **MTissueMainCellNumDC**, the central hub tissue of the synchronisation (all tissues get their names from the **DC** that creates them) consists of one instance of **mainCellNumSyncCell** and one instance of **SpecialBlooDCell** that performs the synchronisation. The **mainCellNumSyncCell** receives changes in the total cell number, and updates its own copy with this change. This copy is actually the same variable as the one that exists inside all the organs and contains the value that can be seen by all the cells inside that organ. After this value is updated (inside the **CellNumSyncOrgan**), this updated value is sent to all the other organs with the **SpecialBlooDCell**.

The other part of the subsystem is the side tissue called **STissueCellNumDC**. This tissue contains one instance of **currCellNumSyncCell** and one instance of **currCellNumChangeProcessingCell**. The **currCellNumSyncCell** receives updated values sent by the **mainCellNumSyncCell** and updates the value inside the local organ. The **currCellNumChangeProcessingCell** detects local cell number changes with the help of a function inside the organ, and sends this change to the **mainCellNumSyncCell** after resetting the change indicating local variable as depicted in Figure 11. The local number of cells changes with the **adDCell**, **die** and **dieIfCan** functions of the tissue. Since the **currCellNumChangeProcessingCell** does not receive jobs, it cannot detect the shutdown and kill signals the usual way. To make sure that this cell dies the same way as others, it only checks if it has received any jobs at all, and since the only way this can happen is with the shutdown job (the kill signal would come at a later time) it dies then. The only problem is that it is sleeping until a cell number change is detected, which must be changed in order for this cell to check for new jobs in its queue. So the local **DC** must change the local number of cells deliberately well after sending the shutdown signal, otherwise the **currCellNumChangeProcessingCell** would sleep till the very end and this would prevent proper shutdown procedures.

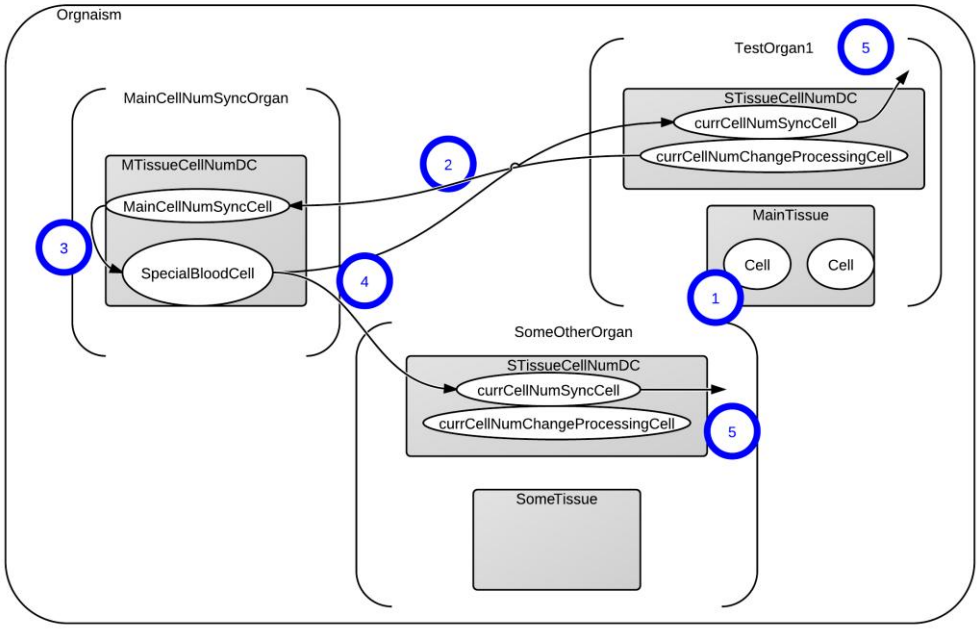


Figure 11 – Cell number synchronisation process.

The process consists of five steps. 1. A cell dies (or is created) 2. The currCellNumChangeProcessingCell reports the change to the main synchronisation tissue 3. The MainCellNumSyncCell registers the change and sends it to the SpecialBlooDCell for circulation 4. The SpecialBlooDCell sends the updated information to all synchronisation support tissues 5. currCellNumSyncCells register the new cell number value in their organs.

The cell number synchronisation subsystem is a good example of both value synchronisation throughout the organism, and the usefulness of the multi-level hierarchy. It also demonstrates unconventional cell shutdown procedures when a cell does not work with jobs but monitors local changes instead.

6.3 Optimisation subsystem

The idea for the optimisation subsystem was to capture the notion of adaptation in nature, as one of the main ideas that drives nature forward is what can be called the long term adaptation called evolution. Sommerhoff (1950) describes the three levels of adaptation with mathematical precision and tools. The long term version that has a time scale greater than an organism's lifespan we call evolution, the short term that happens within minutes and coordinates perception and action (Johnston & Turvey, 1980), and the mid-term being almost everything in the middle we call learning.

Adaptation in the VLO however cannot work fully like in nature, partly because we do not use the VLO long enough to be able to speak of many lifespans and organisms to create an optimal or even useful evolution, and partly because there are many entities and time scales involved. It is easy to see that there are two major time-scales present, one that has the organism's lifespan and the entities sharing these are the organism, the organs and the tissues, the other scale is the lifespan of a single cell. There are more than enough cells to be worth looking at adaptations on this scale. Both of these scales can adapt within their lifetimes, throughout what can be called a population and even short-term adaptation has a few examples.

Short term adaptation happens for example with cell self-replication when there is a spike in the incoming workload. Mid-term adaptation can also be seen in self-replication as each cell is refining a few variables which try to optimise the whole process. These two scales can be implemented in any cell for whatever reason easily, and similar approaches are present in all aspects of programming in general as a feature of self-optimisation and learning, may it be just some information caching or any method seen with costly operations. The long-term adaptation is however something that has to have a centralised support on the long run. Whenever a new organism is born, it always starts with no experience at all. All of its cells are blank and identical. Similarly, when a cell self-replicates it creates a new instance of its type from the blueprints. This

method is impractical as the new one might have so much less information than the parent that it performs several times worse. So the idea was to save the knowledge of any dying cell together with some rudimentary fitness value that the cell itself determines, and do some version of crossover and mutation on the data where possible and allowed, and give the new data from the pool of knowledge to the new generation of cells. If the pool of knowledge were saved to a persistent location, then this information could be inherited by the new cells of the next generation, and some form of evolution would be implemented. The basic idea was that every cell would determine on its own (i.e.; by the programmer) what information would be inheritable, and every cell type would provide all the functions to handle the data, so from the optimisation subsystem's point of view, this data would be both opaque and uniform. The handling of this information would of course carry a considerable amount of overhead, so it would only be practical to use this information and method where the blank cell would perform considerably worse than one with experience. The current cells are basic enough not to meet these criteria so this subsystem is disabled in the current builds due to speed issues because of the overhead. This subsystem should only be included in more complex organisms.

The side tissue **STissueOptDC** should be present in all organs planning to use the optimisation feature. It only has a **DC** inside, and it is a special kind of **DC** as it does not create cells in its own tissue, but in other tissues inside the organ. Specifically, it creates one instance of **OptimisationCell** for (almost) every type of cell in every tissue in the organ. Normally tissue **DC**s create cells inside their own tissues, but it could create a cell that first goes through every tissue in the organ and sticks with one cell type that does not have any **OptimisationCell** attached to it, and this is repeated until the first **OptimisationCell** arrives back, but that would make the code unnecessarily complex so as long as this optimisation does not cause any problems with the emulation it will remain in effect.

The **OptimisationCell** has the responsibility of maintaining enough new knowledge (or genotype) of each cell type so that when a new cell is born it can

use one from the pool. The idea is that there are two knowledge pools for each type of cell, the **inactive pool** that has the knowledge of dead cells, and the **active pool** data that has ready-made knowledge for the new generation (Figure 12).

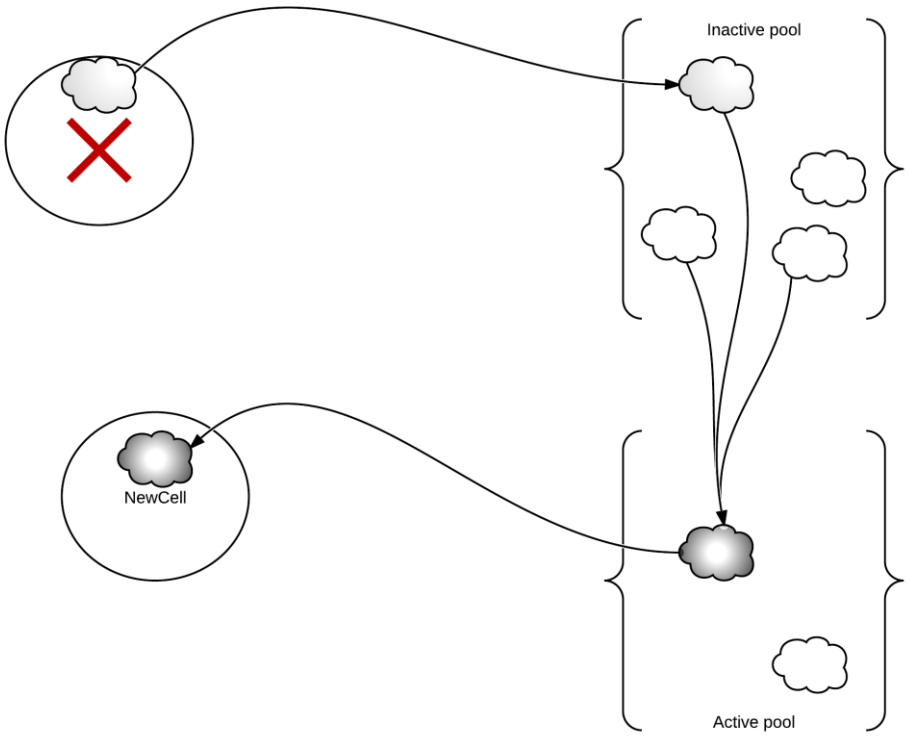


Figure 12 – Saving cells' knowledge.

Dead cells' knowledge is copied into the inactive pool; several of these are combined into new knowledge packs that can be copied into new-born cells.

The **OptimisationCell**'s job is to do selection from the **inactive pool**, combine the selected knowledge data and do some kind of mutation on it, while not knowing anything about what the knowledge means or even contains. For this

reason, each cell type that wishes to use this optimisation has to provide functions to use with its knowledge data by the **OptimisationCell**. These functions are stored with static member function pointers that define the following operations (applied to the knowledge structure):

- Number of fields inside the structure
- Unpacking one field indicated by its index
- Unpacking the mutation function pointer from the structure for the field indicated by its index
- Destructor function pointer for the structure
- Constructor function pointer for the structure
- Packing a field into the structure indicated by the field index

The **OptimisationCell** first sets up its environment and determines all the functions it will need to use. Then it waits for the **active pool** to have less than ten elements or less than the cell population of its type divided by ten. These are arbitrary values that can be further optimised in the future. Normally at the start the **active pool** is empty, so the cell will not wait at all, but if the values will be saved and loaded from a persistent storage between organisms then the pool can have data inside. This cell, just like the **CurrCellNumChangeProcessingCell** does not have any jobs and has a main loop that waits for the **active pool** to change size (actually waiting for a signal to be emitted that is connected with the size change), so the local **DC** has to emit this signal after sending the **shutdown signal** to this cell type.

When there are not enough elements in the **active pool**, it extends it with as many elements as needed by first constructing a new element, then with every data of the knowledge structure it selects a candidate from the **inactive pool** with a probability that is determined by its fitness value, it applies the mutation function that includes the probability of mutation as well, and packs it into the

new knowledge structure. After all elements of the structure are created, it inserts the new knowledge into the **active pool**.

When the **active pool** is filled above its minimum size, the **inactive pool** is cleaned so it does not have excessively large amount of knowledge inside it. It is kept above a hundred elements and below the number of cells multiplied by ten. If the pool has too many elements, one element is selected with probability based on the fitness values and this element is destructed part by part to free the memory.

This optimisation system uses parts of evolutionary algorithms. It defines fitness values to individuals, gene-like parts tha define parameters (variables) of a cell. Depending on how a cell type's optimisation is defined by the programmer, the cell's optimisation routine can use crossover or mutation on these genes with the operators provided by the cell class. The pools provide a buffer for the algorithm to work optimally.

This shows that the optimisation subsystem has to perform much maintenance to do with high level of information hiding so this makes it only worth using if the knowledge of individual cells are important enough. Otherwise parts of the system can be used, for example just selecting one knowledge element from the **inactive pool** to be transferred to a new-born cell without any crossover or mutation that might speed new cells up but would be very cheap to achieve, although this version limits versatility and thus can prove to be far less optimal.

7 Sample organisms

Sample organisms are amongst the simplest ones that demonstrate how to create an organism with the framework. The Sample System is one that is included in the framework that demonstrates how to create a skeleton and do the most basic functions. The Test System is an extended version of the Sample System that has been used throughout development to test various functionalities of the framework. Of course this organism was constantly modified to test these functionalities; the one described here consists only of the main function without any specifics and was later the base for the cancer emulation.

7.1 Sample system

The sample system acts as a tutorial inside the framework to show how an organism should look. The main **DC** is called **SampleOrganismDivineCell** (30 lines of code ,excluding a few lines of definitions in the header) and all it does is to create two organs, the **SampleOrgan** and the **CellNumSyncOrgan**, and call all the default functions written for organism **DC**s that have no special needs.

The **SampleOrgan** (9 lines of code) creates a **STissueCellNumDC** support tissue and a **SampleTissueDC** main tissue and sets up their properties and relations.

The **SampleTissueDC** (31 lines of code) creates two cells, **SampleCell** and **SampleCell2**, sets them up and runs the default functions of tissue **DC**s.

SampleCell (38 lines of code) generates numbers from 0 to 20, sends them to **SampleCell2** (45 lines of code altogether) with result path included, waits for the results and logs the whole process. **SampleCell2** waits for jobs, sends back the number doubled as results and logs the process. **SampleCell** generates the shutdown signal in the end for the organism, **SampleCell2** uses the default shutdown routine.

In essence, the sample system sets up almost the minimal number of objects of a correct organism, but also shows how the cell number synchronisation subsystem and the logging subsystem should be inserted into the organism. The sample cells show a minimum amount of message handling, but as can be seen in Chapter 8, this structure can give enough complexity to study the cancer behaviour.

7.2 Test system

As mentioned before, this organism is the body of the cancer emulation, and was modified many times during the development process to test various parts of the framework. Knowing how this organism is built up and work may help the reader better understand the cancer emulation described in the next chapter.

The main **DivineCell** for the organism is called **TO1DivineCell**, and creates the usual **CellNumSyncOrgan** and the organism's only custom organ, **TOrgan1**. **TOrgan1** sets up the support tissue of cell number synchronisation and a main tissue named **TT1DivineCell**. The tissue **DC** has three types of cells, one instance of **TestCell1**, one instance of **TestCell2** and initially two instances of **TestCell3**. It is worth noting that **TestCell2** does not have any shutdown sequences as it is a factory type of cell, its input drives the whole organism and this cell initiates the shutdown for the organism.

As **TestCell2** is the factory for data, this cell is detailed first. The cell's purpose is to send bursts of jobs to other cells so a fluctuating workload can test the self-replication process, the optimisation process and in turn the cancer emulation by constantly making cells self-replicate and die. The algorithm includes nested loops where massive amount of jobs are being sent and then a wait so the jobs can be processed and the cell count returns to a lower level. One example setup sends 1000 jobs, then waits 10 s, then sends 10 lots of 100 jobs followed by a 3 s wait time, then 50 lots of 1000 jobs followed by a 3 s wait time, and repeats this 50 times. Altogether it sends 50×(1000+(10×100)+(50×1000)) = 2 600 000 jobs. The wait times are included to make sure that the cells can process the jobs, and this can be logged. By far the most costly operation is the

logging to the screen, and this happens long after the cells have finished processing maybe even the whole batch, so the wait times have to be big enough so the operator can actually see the wait. The middle section of 10×100 jobs are sent because this works best to decrease the cell count as every cell that processes a job in this batch will see that there are too many cells working and will die in the end. But since there might be a delay during which they are allowed to enter the maintenance process, 100×10 works much better than 1000 in one batch (Figure 13). The jobs consist of a number that represents where the process is at the moment, and a string that is different in each of the three parts to help monitoring the situation. All the jobs are being sent to **TestCell1**. After all the loops have exited, it sends the **shutdown signal** to the main **DC**.

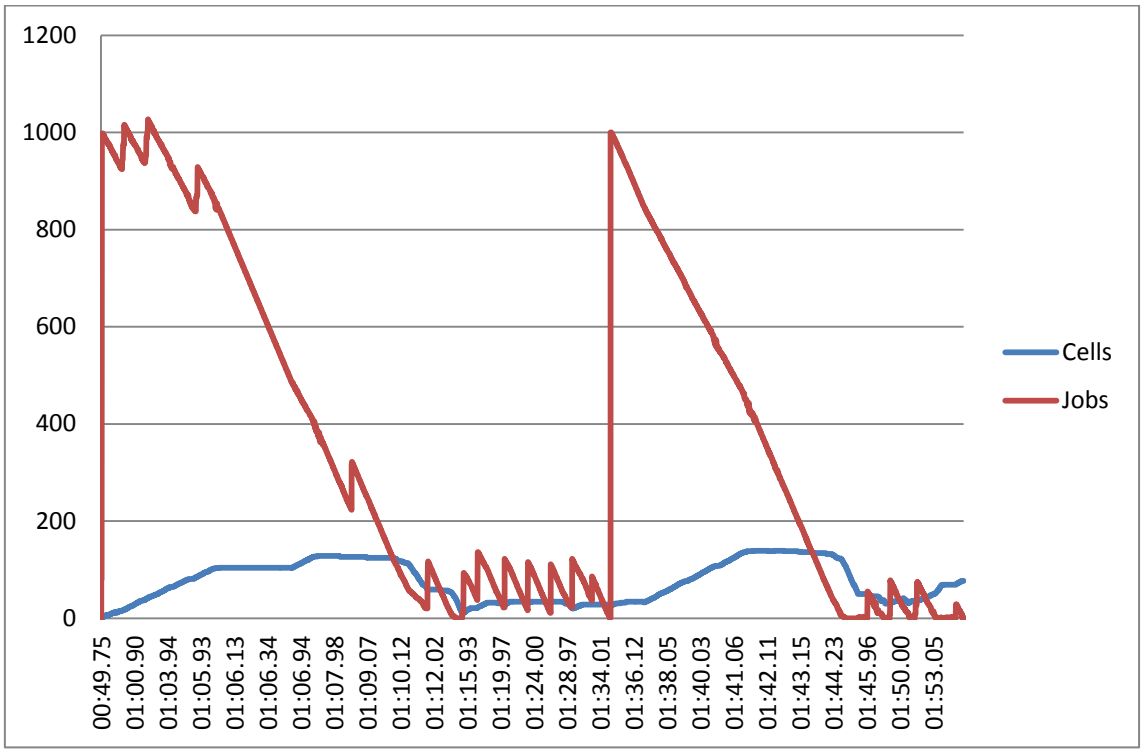


Figure 13 – Cell number responses to fluctuating workload in a cancer emulation.

Note that the cancer emulation was stochastic enough to change the bursts of jobs waiting in the target cell's queue from the original

programmed values, but it can be seen how the cell number change is able to cope with the workload change.

TestCell1 is the one that is capable of developing aneuploidy in the cancer emulation. This cell logs any incoming jobs, and sends the received number twice in one job (as in x,x) along to **TestCell3**. It waits for a result from the job request, which is a number and logs this result. It continues to do this until the **shutdown signal** is received, after that the default shutdown procedure follows.

TestCell3 receives the jobs from **TestCell1**, adds the two numbers together and sends the results back; effectively **TestCell1** will receive back the doubled number. The shutdown procedure is the default one.

The main function of this organism makes it a good candidate to any kind of tests where the self-replication and cell death have an important role, which is why this is the base for the cancer emulation.

8 Demonstration: Emulating cancer with a virtual living organism

8.1 Goals

The Virtual living organism (VLO) library makes rapid prototyping possible in many fields of biology. To test its effectiveness, cancer was chosen to be prototyped. The goal was to create an effective emulation of cancer reusing existing materials (e.g., cells, tissues, organs, organisms) to create virtual emulations of both benign and malignant tumours.

8.2 Chromosomes

Cancer in the VLO is emulated using the VLO programming library by emulating aneuploidy, the most likely cause of cancer (Duesberg, 2007); (Fabarius, Li, Yerganian, Hehlmann, & Duesberg, 2008). Aneuploid is defined by the Merriam-Webster dictionary as "having or being a chromosome number that is not an exact multiple of the usually haploid number". As chromosomes define what a cell is made up of and how it works, it is a logical choice to define a virtual chromosome as a closely coupled small block of code the cell executes. In this case, the cell will execute the chromosomes in a predefined order. Later this definition will allow extending the scale of the cells by defining genes as parameters inside the chromosome code at first, and as even the types of commands later on.

8.3 Aneuploidy

Aneuploidy then can be modelled as any chromosome block individually can be left out of the cell's programming or it can be run multiple times. For the sake of this initial emulation, the order in which these blocks are executed cannot change, so multiple instances of the same block are executed together one after the other, but the coding allows this to change if needed.

Having the program of the cells broken up into blocks that might be executed more than once or not at all gives some interesting design challenges. The easiest to resolve is the data handled in these blocks. If the original program of the cell is changed it can easily happen that a block tries to use undefined data. This would force the operating system to shut the whole VLO down, which is unneeded as in nature there is no aneuploidy that instantly kills the whole organism hosting the colony of aneuploid cells. Extra checks for these cases are incorporated into each block.

A harder challenge comes from the control flow commands as these need to be disassembled into many blocks. A simple conditional statement consists of a condition block, a "then" block, and on "else" block and the end of statement block. If any of these are missing, the flow of the program is altered (Figure 14). For example, if the condition is missing, only the "then" block will be executed all the time. If the "then" or "else" part is missing, the other will be executed all the time. If the statement ending block is missing, the "else" block will be executed every time after the "then" block if the condition was true. The broken up versions of these control flow statements work with jumping commands that search for the next or previous instance of the given block ID and continue the execution from there. If they don't find any such block, the jump command will be ignored and the execution continues with the next statement in the line of statements given by the chromosome list.

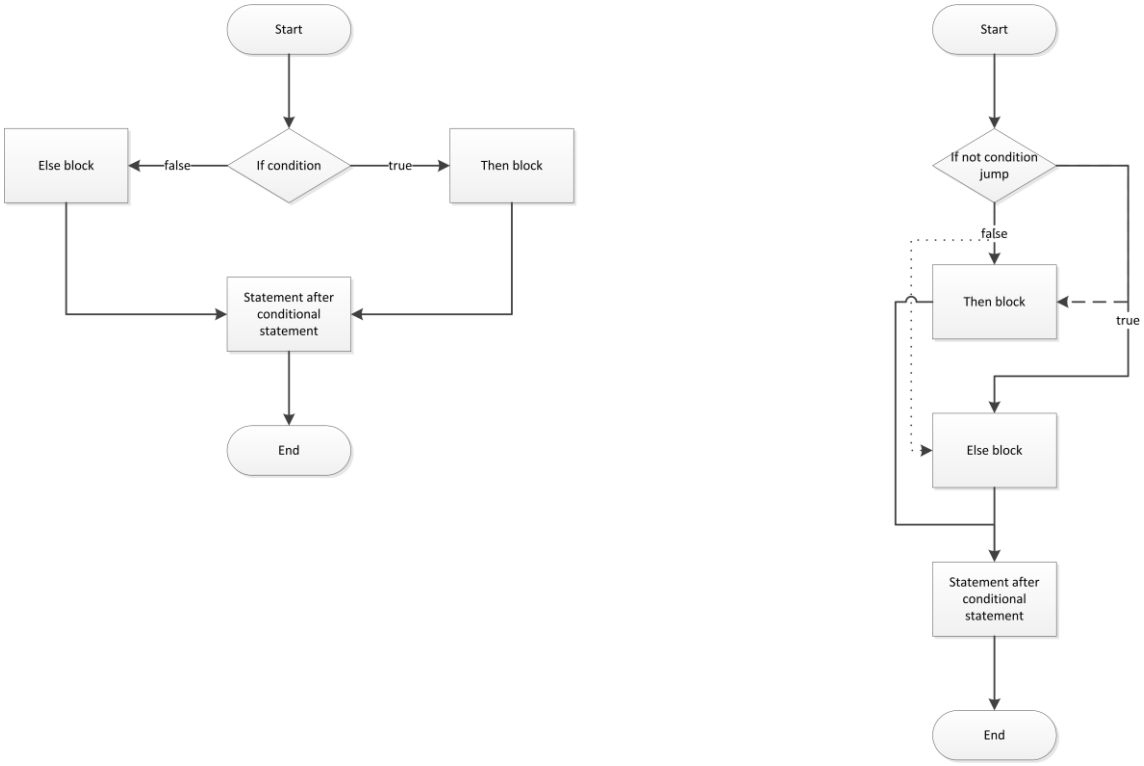


Figure 14 – Standard and chromosomal versions of a conditional statement.

The standard version of the “if” statement (left) is independent of the arrangement of the blocks. The chromosomal version (right) is more sequential. If the destination of a jump command is missing, an alternate jump will be performed to the next block. If the "Then block" is missing, the statement will continue to the "Else block" regardless of the condition (dotted line). If the "Else block" is missing, then the "Then block" will be executed (dashed line).

Multiplying or leaving out a chromosome can have multiple effects. The direct effect comes from the code inside the actual block. Having a chromosome more than the intended times can only have this effect because of the local property of the multiplication (the order will not change). The indirect effect comes from the control flow changes inside the program if a critical part is missing. For example the direct effect can be that the left out chromosome will not increase

the value of a variable, the indirect effect could be that it will not jump to another location because of a condition, so the flow of the program changes entirely. Having a crucial block multiplied ensures that it is more unlikely that it will be left out in the future.

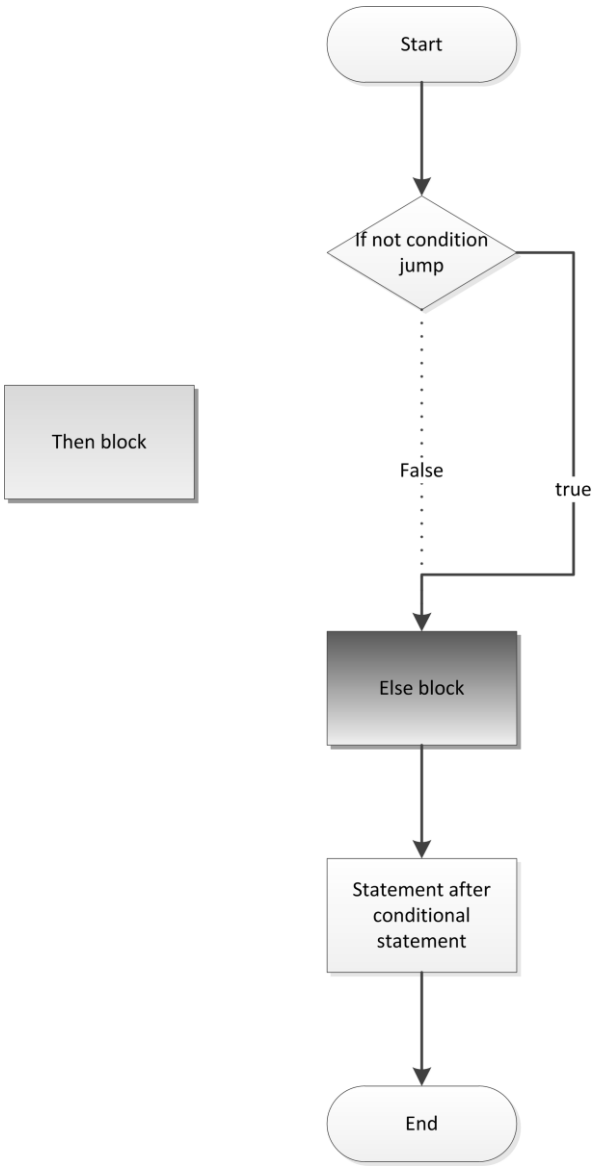


Figure 15 – Direct and indirect effects of a missing chromosome.

If the chromosome that corresponds to the "Then block" is missing (dark block) then the direct effect of this would be the contents of the block itself. The indirect effect would be the change in the control flow, the definite execution of the "Else block" (darker block).

Copying the chromosomes is done during self-replication. An error is the copying is introduced with a ±1 change in the number of the current chromosome being copied and a probability that will on average introduce 1 error in 3 self-replications in the current setup. This probability ratio can be changed. After the copying there is a check that compares the n^{th} chromosome in the copy and the original with one random number n. This part can of course be multiplied in future generations resulting in a more stable chromosome pool.

This cancer emulation might look somewhat similar to genetic programming (Koza, 1992), (Koza, 1994), as it generates a program code with a stochastic algorithm, although this does not aim for anything, and is very different in its core details. For example the cancer emulation does not create code that describes how to make something but the final product that is executed. Also there is no crossover or mutation in its strictest sense, only multiplying or deleting of parts.

8.4 Results

Out of 24 chromosomes in the test system leaving out any of 8 can cause death of the cell, leaving out any one of 6 can cause longer life for a cell, multiplying 3 can cause death and one can cause longer life. Primary function is only affected by 2 chromosomes and self-replication by 5. For a short description of chromosomes and the direct effects of aneuploidy, see Table 5. The top 50 aneuploid cell types can be seen in Figure 16.

Chr num	Description	Jumps to	Effect of omission	Effect of multiplication
1	Wait for jobs	2	Dies immediately	Discards some jobs
2	Check for kill signal	26 if job contains kill signal, 3 if not	Potential immortality, discards kill signal	No change
3	Check for shutdown signal	4 if shutdown signal detected, 5 if not	Dies prematurely	No change
4	Die if not the last cell of its kind	8 if last cell of its kind	Potential immortality, discards shutdown signal	No change
5	Process job	6	Skips job processing	No change
6	Go to maintenance if not last cell of its kind	10	Skips maintenance	Multiple maintenance
7	Random death	26 if dies, 8 if not	Lives longer	Lives shorter

8	End of main loop	1	Dies prematurely	No change
9	Skip maintenance	26	undefined	No change
10	Check if maintenance should be skipped	7 or 8 if 7 does not exist	Maintenance happens more often	Maintenance happens less often
11	Get maintenance data, try to enter maintenance cycle	13 if successful, 12 if not	Maintenance will not work	No change
12	Set optimal access time variable	23	No change	Maintenance happens less often
13	Change last modification variable	22	Minor change	No change
14	Get workload data	18	Always multiplies	No change
15	Chromosome copy process	16	No aneuploidy	No change
16	Aneuploidy check	15 if aneuploidy found, 17 if not	More aneuploidy	Less aneuploidy
17	Self-replication	21	No multiplication	Rapid multiplication

18	Check if it should die	19 if it should, 20 if not	Dies always (or self-replicates instead of death)	Dies less often
19	Die if not last of its kind	21 if last of its kind	No death	No change
20	Set optimal modification time	21	Minor change	Minor change
21	Set other maintenance data	22	Minor change	No change
22	Set optimal access time	23	Minor change	Minor change
23	Set last access time	7 or 8 if 7 does not exist	May die after maintenance	No change
24	Cell movement checks	7 or 8 if 7 does not exist	No cell movement	No change
25	Cell movement	7 or 8 if 7 does not exist	No cell movement	No change
26	Death	-	undefined	No change

Table 5 – Chromosome descriptions and direct effects of aneuploidy

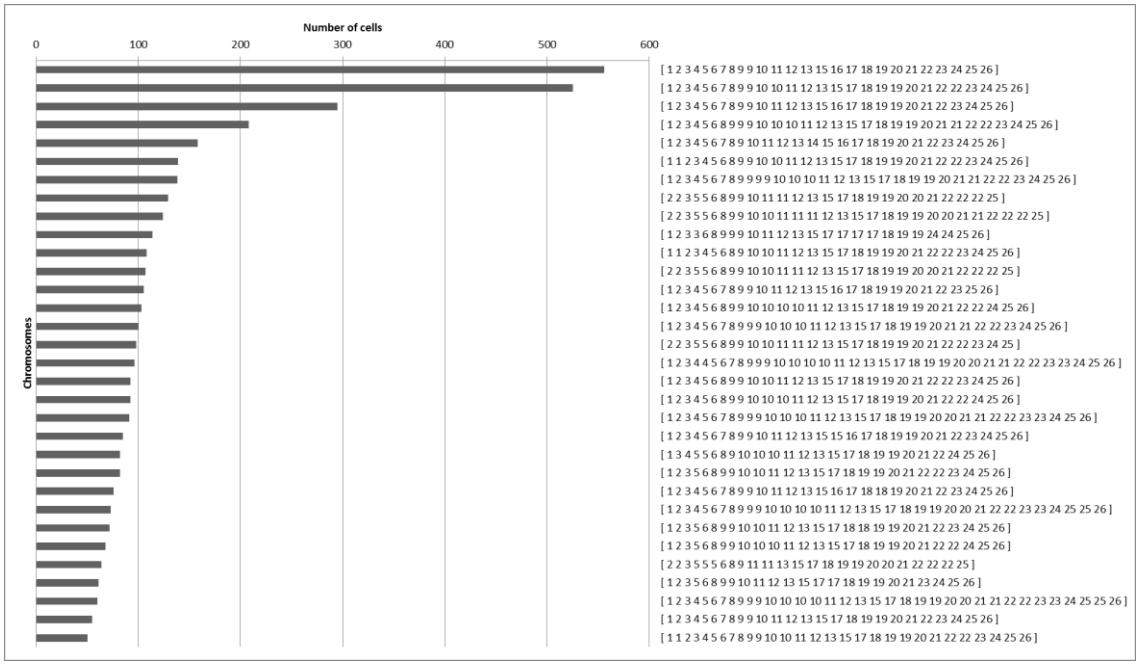


Figure 16 – Distribution of aneuploid cell types within a cancer emulation.

(chromosome combinations with at least 50 instances) It can be seen that the most dominant ones have chromosome 14 left out, which resulted in the multiplication of the cell in its every life cycle. This resulted in these cell types becoming the most dominant ones.

There are also effects that can only be observed with a certain combination of chromosomes missing and multiplied, like absolute immortality or even more death or self-replication modifying effects.

At first, only a tumour that can be called benign was programmed. A far more aggressive variant, that can be classified as malignant emerged during testing without any intention of creating it. A combination of chromosome changes resulted in a shift in the cell cycle which normally has a pattern of ABCDEABCDEAB... to ABCDCDCD... with A being the wait for a new job, B the processing of the job, C the call of the maintenance function, D the maintenance itself and E getting a new job and jumping back to A. The second variant does not wait for any job and as a result runs constantly without any delay and what is more it runs the maintenance routine all the time. It is a small

step from here to introduce one of the several modifications that enables immortality or a very rapid self-replication to make this type the dominating cell in terms of hogging the CPU.

The emergence of this malignant tumour cell changed the behaviour of the cancer simulation all the time. With the introduced pauses that are keeping the cell numbers in a controlled range the benign simulations used the CPU of the test machine at about 6%, while the appearance of the malignant cancer cells boosted this up to 100% immediately, coupled with a quick boost in the number of cells.

The main difference between the virtual versions of a malignant cancer cell and benign cancer cell is the loss of local resource dependence. While immortality can be achieved many ways even if immortality refers to not only an individual cell but the cell population, like replicating many times before death or disrupting the death process (apoptosis), the idea to bear in mind is that the numbers of active cells, and even the cell's activeness is regulated by the number of jobs waiting in the tissue's queue. This results in that if a cell moves to another tissue where it is not supposed to be, normally it will be inactive, because it conserves resources until it has jobs to do in a similar form to hibernation. Until this step is disabled, any cells moved to a place where there will be no jobs for it will remain hibernated until the organism dies. In the current suggested structure of cells, this mechanism can only be disabled by an alteration in the control mechanism of the cell, a chromosome that describes the order of things to do, not one that defines how an activity can be done. Even if this chromosome gets altered in addition to the one that enables the cell to lose anchorage and be moved to another tissue (implemented later as part of metastasis), it could die without an effective immortality, as other chromosomes could kill the cell if they detect that it is not required at its new tissue. As a result, multiple chromosome alterations are required to achieve malignancy, with the absolute necessity of control flow alterations. The actual probability of an alteration that results in malignancy depends on the original cell's structure,

function and implementation, and thus can be customised to reflect any probability that the experiment requires (e.g., real world experimental results).

As a next step to create a truly malignant cancer, invasion and metastasis was programmed. These features were introduced by a normally inactive code that was based on the relocation code of the blood cells. The result is that any type of cancer can move to the nearest tissue by enabling the code block, but as their cells will not get any jobs there, they will be inactive for the time being. But if a malignant cancer cell is transferred it will not wait for anything and is free to stay and multiply or continue to other tissues or even organs. During the tests, several invasions and metastases were observed. These malignant versions have many of the growth characteristics of malignant cells in biology, like immortality, decreased requirement of growth factors and even loss of anchorage dependence (Ruddon, 2007).

The cancer test VLO functionally consists of 3 types of cells, similarly to the test system described in the previous chapter. These cells are: the factory cell that generates numbers and the cancer-prone cell that multiplies (multiplexes) those numbers and sends them to the third type of cell and logs the results when they arrive. The third cell adds the numbers together, thus doubles the initial number and sends it back to the cell it received the job from. This setup gives an easily controllable environment that has multiple types of inter-cellular communications and also generates logs that makes it easy to see what steps were skipped or multiplied. It also results in a VLO that does everything a living organism should do.

There are also lots of helper cells that are found in most of the virtual organisms like divine cells that help create and maintain tissues, synchronisation cells used at creation and death, maintenance and optimisation related cell types used throughout the life of the organism.

8.5 Conclusion

Creation of the virtual cancer took me less than a week in terms of designing and coding. Aside from showing that reuse of past results and a small amount of work can result in a viable emulation of a complex biological phenomenon it also showed that the emulation can be “creative” or at least unexpected enough to give insights into how things might work in related parts of biology. With the fast development times and even faster simulation execution times (which in this case were a matter of minutes, about half an hour on average on a desktop computer), this system could be a tool to be used before laboratory tests as a proof of concept and first check of theories. This work is, to my knowledge, the first instance of emulation of cancer as a result of aneuploidy, and it shows how this framework can be used as a test bed for further studies in many areas of biology and software engineering.

This cancer emulation shows a way emulation and VLO can alter the models used at the creation of drugs. For example using this emulation it can be determined what degree of aneuploidy, what number of chromosomal changes is required to increase the effectiveness of cancer. This information can be used by the drug industry for new ways to create detection and treatments.

9 Further discussion and future work

The main discussions of this thesis showed how this framework compares to biology and where and why it diverges. Together with the examples we have demonstrated how the framework can be used and thus completed an initial goal of creating a useable rapid prototyping tool. There are many aspects, limits and possibilities, which are not part of its basic operation and worth exploring. These aspects are discussed in this chapter in the hope of enhancing the usefulness of the framework and by showing even more possibilities where this emulation approach (as well as the concrete structure of the framework) can outshine any possible alternatives.

9.1 Gradual refinement approach of emulation

By its nature, emulation is a top-down approach. While simulations create many small blocks and mainly rules that gradually create a coherent system that is close to what it is supposed to be simulating, emulation creates the biggest possible blocks to create a backbone that is similar to the goal of the emulation, and exchanges these blocks to more detailed and granular versions, but only if needed. The power of the emulation shows that some parts can be much less granular than others just because they play little or no role in the studied phenomenon. This is similar to what happens in manufacturing, when a vehicle is constructed it can easily have micro- or even nanomachines inside that are constructed with appropriate technology and detail, while other parts like the chassis are constructed with far less precise tools. If all parts of the vehicle were created with nano-precision tools it would no doubt increase some performance or quality figures, but not as much as it would increase the cost of production. There are some areas where precision is the topmost priority and other areas where it can be balanced with costs and these latter ones are where emulation can be used far more efficiently.

It is evident that with its current set of tools and building blocks the VLO framework would be inadequate to emulate for example the brain or maybe

even a full and detailed cardiovascular system of a human, but this is because it is not supposed to. As it was demonstrated, the current level and building blocks are more than enough to create an emulation of cancer, and there was no need to create many more building blocks at this stage. This is an easily extensible framework that shows how to think in a biological emulation-like manner and gives tools to realise these thoughts.

The framework is deliberately designed to be modifiable, even at its core, to be able to fulfil any future needs. There are several layers in its core operation that have well defined interfaces that hide the inner working for this same reason. If, for example, it was necessary to implement a more precise emulation of the circulatory and message delivery system, it could easily be done, and if it has the same input parameters as the old one (namely the address and the message, without which a message would be meaningless anyway), then the change would be undetectable by the users of the service, the cells.

Of course this process works in both directions. If the farthest possible region from the core's point of view needs to be more precise, it would not affect any other parts. For example if one organ or tissue or even just a cell type would require either spatiality or a physics simulation to work accurately enough for the experiment's needs, it can be implemented as something that uses the internal properties of these parts, without any other part of the system noticing any difference. This shows how different spatial and complexity scales could work together in one emulation.

But what about time scales? Biology has many fields where in one experiment there need to be many time scales observed (Hunter & Borg, 2003), where one part does thousands of things while any other would only do one in the same time. An effective emulation tool would need to be able to emulate this. The solution in this case depends on the actual problem at hand. In the easiest case, this is done automatically. If the fast moving part has thousand times more work to be done, its workload can trigger enough self-replications to balance the system (as it is the case with biological immune and nervous systems (Dasgupta D. , 1997)) and then there is no need for any manual

adjustments. If this does not happen because for example this part (let us assume it is a cell) does not have any incoming jobs, then it can be given a custom maintenance process that ensures its relatively superior numbers within the organism, for example it keeps the cell count at a predetermined percentage of the total cell count of the organism. This will ensure that whatever this part does is done with greater priority, so it is effectively in a faster time scale.

Normally the relative number of cells would give enough priority for a certain type of cell to be able to work even on a different time scale compared to others, but if for any reason this would still not be enough it is also possible to include one of many types of synchronisation to ensure absolute time scale precision. For example all fast cells could emit a signal after doing one “job”, even if it is not a physical processing of a job request, and an outside tissue could count these signals and start the operation of the rest of the organism only when the predetermined number of signals have arrived. If a fast paced cell were to be added to the example called test system, the synchronisation tissue (or cell) would signal to the factory type cell that it can emit one more job to the rest of the system only when the new cell type did, for example, 10 main loop cycles. If even this would not be enough to achieve the time scale difference, a standard synchronisation line could be added to all of the cells at their main loop (or even maintenance routine) that starts the loop by sleeping until the synchronisation signal arrives.

These examples show how different levels of precision can be achieved with increasingly more direct and complex changes at almost any level of the hierarchy (Table 6). Emulation can provide a tool that fits the exact criteria of the experiment, and the VLO framework can provide a rapid prototyping tool that can be used to create a customised emulation for almost any experiment with a fraction of the work requirement that would be needed to create a custom simulation. What might be even more important is that the speed of emulation can be many orders of magnitude faster than a simulation equivalent. This enables to run far more emulations creating massive statistical data or in some

situations it would even enable almost or even exactly real-time emulations that can have practical uses in medicine.

Hierarchical level of change	Change	Result
No direct change	No direct change	Let the self-replication function handle time-scales automatically
Cellular level	Change maintenance function	Artificially increase cell numbers
Tissue/Organ level	Synchronisation subsystem	Keep other cells inactive forcefully

Table 6 – Possible solutions to simulate different time-scales at different levels of the hierarchical structure.

9.2 Limits and possibilities in terms of spatiality

The problem of spatiality includes lots of limitations and that is why the current model is based on a hierarchical structure. Spatiality includes the problem of detail by its continuous nature, and even this problem is solved many ways in simulations and emulations. For example most of the cellular automata use discrete spatial scales (Wolfram, 1983) as each cell resides in a two or three dimensional lattice. But there are also examples where a discrete spatial scale is not enough, for example when cellular surfaces and shapes are important parts of a simulation. The solutions for a continuous spatial system are computationally heavy, and thus it would be unwise to make such system the base of the VLO framework. Equally it would be counterproductive to make discrete special orientation the base of the framework as if continuous spatiality is important for parts of an emulation it is harder to superimpose a continuous

scale over a discrete one when they require completely different methods, abstraction and way of thinking.

In fact, the current hierarchical structure has an element that is similar in functionally to a discrete lattice-based spatial system. The lattice defines neighbour properties for each element, and similarly there is a next in line property for each tissue and organ inside the organism that creates the circulatory system. Even if this is only for one dimension and the current implementation is only unidirectional, the same method could be implemented for cells where needed and thus creating a lattice based spatial system (Figure 17). However this solution would not be without disadvantages. It is perfectly suited for fixed elements as tissues and organs. But if elements are constantly changing; it is not nearly an optimal solution. In one dimension, it is easy to insert or delete a cell from any position, but in two dimensions the extra dimension has to be adjusted as well. For example if we think of a lattice where one element is deleted, the adjacent ones' pointers need to be adjusted to skip the deleted element, but how is it stored that there is a free space? If a new element can be inserted into any position between any two elements, it would not create a valid two dimensional lattice, free spaces have to be stored somehow, and if there are dummy objects, this solution is far from being optimal (again, the lattice itself is not stored, only the links in cell that point to the next and previous elements create the virtual lattice). Even so, this solution is probably the best with its least amount of interference with other functions if spatiality is only needed in one part of the system and its objects are stationary.

If not, there is also the possibility to store the lattice externally (for example inside the tissue), and the elements know their position inside this, and the lattice (matrix) is made of pointers to its elements. If the task required a physics simulation working with the spatial arrangement of cells this external storage of the spatial matrix would be the best choice. One of the disadvantages of this solution however is that all the information is stored in one structure, and the most optimal solution regarding access time is a rather simple matrix. As there are many threads accessing this information at once, updating it requires a lock

so other threads cannot access half-updated data by accident. If the data structure is a really simple matrix with pointers as data values, then it can be only locked all at once, meaning that while an update takes place no other thread is allowed to read, even if it were to read an unaffected region.

The previous solution where location data was stored in cells made this problem easier as locks can be made per cell, so one modification allows all other data to be read at the same time. If the matrix needs constant editing, it either needs to be partitioned so there are several regions that can be locked independently when editing data in them, or having a per element lock. In the latter case all elements would be structures with data and lock inside them, but this would also mean another step to access the data itself which would slow the execution down if this spatial information is heavily used. It is important to note that the discrete timings of the cellular automaton make this process much easier as cells inside them know when it is safe to modify the information and do not have to worry about inconsistent data when reading.

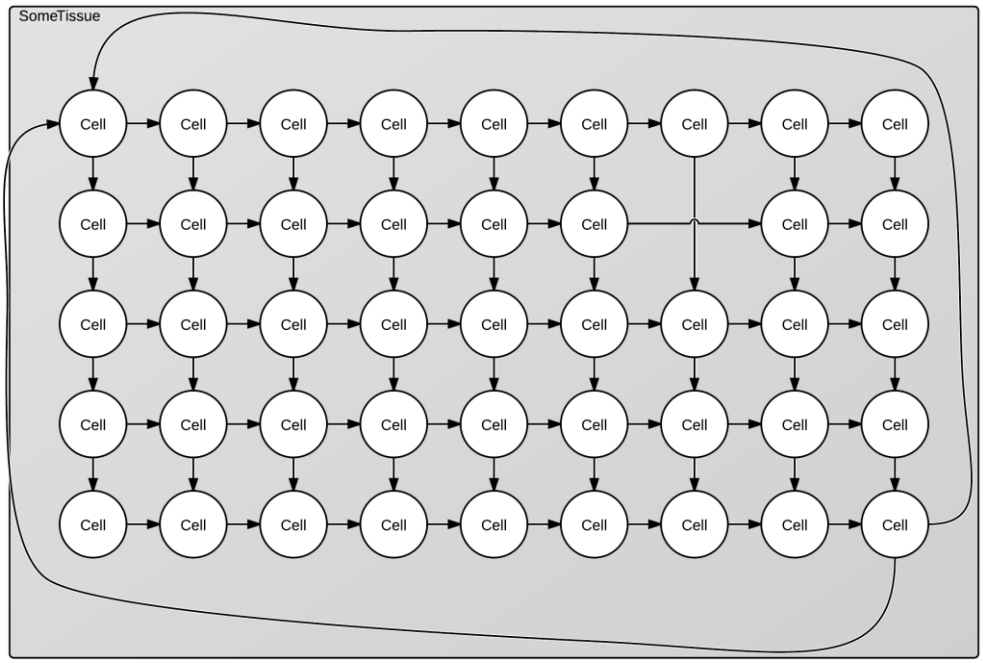


Figure 17 – Example of one possible two dimensional lattice based spatial system with unidirectional pointers

As shown before there are many different solutions that can be optimal for all the different requirements experiments can have, and no doubt if the framework is used there will be some solutions that can be used just as easily as reused tissues or organs. In fact, it would be wise to create any solution as a per tissue or per organ subsystem so it can be bundled within the only region where it is necessary and can be reused more easily in another unit later on. It is easy to see that the solutions where the spatial coordinates are stored inside the cell do not require any outside help, meaning that it they are only used to represent the orientation and relations of one cell type only in one tissue, then this solution can be automatically included in any tissue when the cell type is inserted in the tissue, but the external storage of spatial information in a matrix requires both cellular and tissue-wise support. In this case each cell that uses this feature needs to be in a tissue that supports it which increases reusability requirements.

9.3 The presence of adaptation and possibilities of evolution

As discussed in Chapter 6.3, the VLO framework includes several versions of what is essentially adaptation and evolution. In fact, if evolution as a cross-specimen adaptation is extended beyond its current definition, then having it applied to cells within an organism makes cells' evolution the same as the organism's adaptation, given the different time scales involved. The basic idea of the optimisation subsystem was to create a version of the mid-term form of adaptation, learning (Sommerhoff, 1950). This form can be best applied to cells given their life-span and the number of cells within the organism's life.

If we would apply a version of evolution to cells, it would not make much difference in simpler VLOs, except with extreme settings like very probable mutations. As a conclusion, we can see adaptation in the cellular level as cells try to optimise their inner variables based on their environments for maximum performance (e.g., self-replication in case of suddenly increased workload) even in the simplest cases, they can utilise learning and teaching with the optimisation subsystem (e.g.: best maintenance variables change during their lifetime and can be the starting values of the next generation), and could even have something similar to evolution, but it would not be beneficial for a simpler VLO because of the overhead.

From the organism's point of view, adaptation happens with the learning and teaching process of cells, as it can better regulate its inner functions for any internal or external changes. The mid-term form would vary on the tasks of the VLO as it might not be suitable for any long-term learning. Although some cells optimisations can be long-term enough to be called learning for the whole organism, there is no subsystem yet that would be able to give this knowledge to the next generation of the VLO. In the classical sense it would not be even possible as VLOs probably will not live along with their offspring, although if the cell optimisation data is saved at the hibernation of a VLO and is loaded again at the birth of the next one, it would accomplish something similar.

As for evolution, it is doubtful that a single VLO created for experiments would have enough generations to have anything resembling evolution, not to mention that in the strictest sense of mutation it would be unadvised to create a VLO that can change its internal structure dramatically and thus give results that might be inaccurate, especially if it is supposed to be a classical computer program created for a predefined task.

If VLOs would ever make alternatives to standard user programs, their numbers would be enough for evolution, but in that case it would be even bigger problem to control their correctness and consistency. A wildly changing database program could erase or invalidate the whole database by mistake, and surely such examples can be thought of in every field where we rely on computers. However there could be some flexibility in the speed of the VLOs. If there is a strong consistency check of the output generated by the program (where it is possible at all), then the program might be allowed to evolve as long as it provides the right results. For example if the output (let us say it is some data in a database) can be defined entirely by a set of rules and constraints, then a separate, non-evolving VLO or program could check every output, and could restore the previous version of the evolving VLO in case of an error, while also restoring the environment to the state before the current run of the VLO. It is an entirely different matter when this evolution could result such an increase that would justify all the redundancies and general overhead, but I suspect it depends strongly on the task of the VLO.

9.4 Extensibility and reusability requirements

In many of the chapters, there are examples of subsystems, services, properties and several different units like cells, tissues and organs. Almost all of these examples were parts of a working system, an actual VLO, many times even parts of many past VLOs. But how does one build a VLO out of these components? Can we just put them all into a single VLO and expect them to work? Can we use multiple component of multiple past VLOs together? The answers to these and similar questions define how reusable and in turn

extensible are the parts of a VLO. The answer is, as can be expected, not so simple. There are many different solutions to similar problems throughout many tools in computer science.

A good solution to the reusability problem is to create well-defined interfaces which the components use to connect to the system, and that define each component's inputs and outputs. The standard communication channels, the addressing, the standard storage solutions (like jobs and maintenance variables) are indeed well defined for any new cell to rely on. So as long as there are no new requirements for a component, there should be no problem of reusability.

But why would there be any new requirements? Nature has many solutions to many problems, and most of these are very different. The same problem is solved quite differently in different species, be it the difference of the skeleton structure in mammals, birds or insects or the outer layer of different body parts of humans or trees. So while there are things that can be found everywhere like the requirement for nutrition for each working cell (which can always at least in part be translated to jobs or CPU time slice in some manner), even the incoming information type can change dramatically between cells from chemical information that jobs represent to neural information. Which differs significantly in its delivery method and properties. So each cell, tissue or organ can expect some kind of support from its environment for its operation. A cell type may require some information storage space in the tissue it resides in, a tissue may require for at least one of its cells the cell number synchronisation subsystem in the organ it resides in, and an organ may require another organ with the main **DC** in it to work correctly (as all organs do in current examples). These requirements must be documented and implemented for the parts inside them to work correctly.

In optimal case a component is as self-contained as possible. If the creator of a component created a cell that processes some sort of information and has no other needs, the cell can be used in a pretty standard environment and can easily be distributed on its own, and be reused without any modification. If

however the component is a complex set of cells that work together and use some extra tissue support, they should be reused together with their tissue. The same applies if the set contains cells that should be put in different tissues then they should be distributed and reused with their organ. The bottom line is that all components are easiest to be reused with the minimal amount of structural elements that makes them the least amount of units at their highest level and includes every requirement that is not to be expected of an average basic VLO to have (Figure 18). This results in self-contained tissues, organs or organ groups.

Of course sometimes not all of the component's services are needed, for example there might be a tissue that is self-contained and only one cell is desired for another tissue under construction. In these cases the only solution is to meet the cell's requirements in the new tissue manually. This can be done either with standard programming tools like inheritance of the new tissue from the original parent tissue of the cell to be reused; this can only be done if that original tissue does not have many features that would be unused. Otherwise the only solution is to manually copy code from the original tissue to the new one, but luckily this is just a last resort and probably can be avoided most of the time. Either way, good documentation of at least the interfaces and requirements of each component is required for good reusability – a situation similar to most other fields of software engineering.

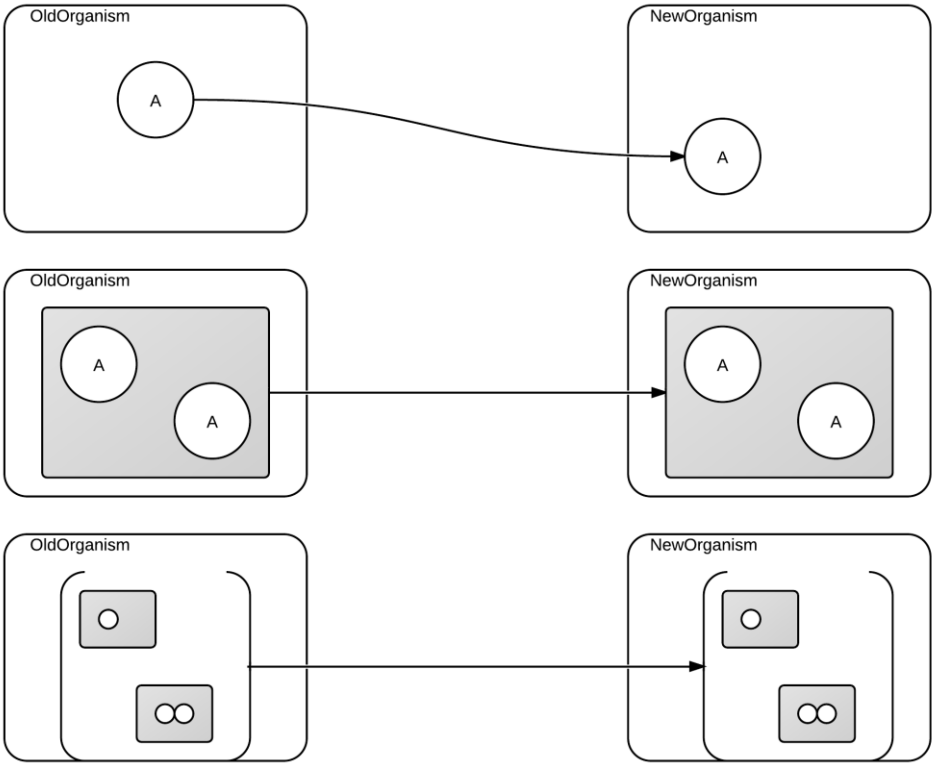


Figure 18 – Reuse of different parts of a virtual living organism.

It is easiest to reuse elements at their highest structural level and not the individual units inside.

Versioning (Gergic, 2003) (as in tracking the different software versions) is an entirely different matter. The current framework does not handle different versions of the same component, although if the version is included in the component name (like **CellTypeA**Ver142) it will be handled as a different cell and thus multiple versions can work simultaneously to serve different parts if they are made for different versions of this component. In the future to increase the usability of different versions of components, it would be best to include support so that each component can declare if it is compatible with earlier versions. The version property would be handled differently from the component

name, and would be included in the addresses. The components would declare which earlier versions are entirely compatible with their current version and all the compatible requests would be delivered to the most recent version of the component. The package that contains each component could include all incompatible past versions so any VLO that utilises them could work with the newest package and use any version they require for their operation. Furthermore each component would declare what other component they require and what version of them, so only the needed components and versions would be included automatically in the actual VLO, and any incompatibilities, unmet requirements and outdated versions would be pointed out automatically at runtime.

The mechanism for this new checking operation would be just an extension of the initial synchronisation process, where even now tissues are reporting all the component types they know and require to work with. If this is extended with the version information, the compatible and current versions of all their components and their components' requirements, then this information can be processed by the main **DC**, sent back for all the tissues to know which versions need to be included. If there is a problem, the main **DC** can report it and halt the VLO before the tissues get the go signal.

The provided solutions and examples in this chapter show that most of the time reusing a component in another VLO can mean that it is enough to include the smallest self-contained version of the new component, but if this would be too big there are alternative and easy solutions to reuse just a smaller part, and describes how can a relatively small extension to the framework make it much easier to be used with different versions of components with mainly automatic processing of optimal component version to be included and the reporting of any unmet component requirements inside the system.

9.5 Scalability, maximum cell number and speed increase options

The issue of scalability was briefly discussed in Chapter 6.2. The current system is quite scalable; it can emulate anything from a single cell organism to a complex organism consisting of vast amount of tissues, organs and cells. In fact the number of tissues and organs is practically unlimited (tens of thousands), although nature does not tend to create vast amount of these inside one creature. The limit is only apparent with the number of cells. At the moment, the maximum number of cells is limited to 10,000 for 64bit systems with not much larger theoretical limit with current threading settings. This number could be increased by a factor of about 2-4, even more with some additional modifications (e.g. stack sizes for threads), but for the purpose of the framework is to enable the creation of the simplest possible organisms tailored for individual experiments.

This task might not even require a fraction of this limit in cell numbers. If it does, then there are several options to increase this. The obvious one is thread pooling (Hyde, 1999). With this technique, the cells that are waiting for incoming jobs or other conditions that are rare enough to make this cell more optimal to "cache out" can be suspended and written to memory with all their data. Their threads can then be reused by other cells that are actively working. This technique is standard practice to reduce the number of threads. In an average VLO, there are several cells that are waiting for an extended amount of time. For example, the main **DivineCell** waits for the shutdown signal from the finishing of the creation process to the end of the VLO's functionality. In this particular case this cell is well worth caching out, although a sleeping thread does not consume excessive amount of resources so as long as this thread limit is not actually limiting, thread pooling is not worth implementing. The real trick with thread pooling is to determine which cells are worth caching out. If a cell's thread has been taken away, it takes some processing to restore it, so if an active cell is cached out it impacts performance.

The best way to determine how active a cell is would probably be to store a timestamp of its last activation, and a subsystem would then scan through cells and cache those out that are inactive for the longest time, but in the meantime try to cache out as little as possible not to impact performance. It would need to monitor the total cell count closely. It would also require the modification of communications as the arrival of new jobs would need to check if there are any active cells and would need to trigger a cell activation routine if needed. Better optimisation method would be to decrease the numbers of a single cell type in the same tissue sooner than the maintenance routine requires it, rather than starting to cache them out. It can happen in the current system that when an increased amount of jobs comes in, the cells multiply several times until they can process the jobs and then all start to sleep when the workload is decreased considerably to a level where no new jobs come in for some time. In this case they do not have the chance to enter maintenance phase and may wait until the next batch of jobs arrives to decrease their numbers if needed. Detecting such situations before caching threads would be required.

There are also solutions to the cell limit problem that would also increase the speed of processing. More cores (usually coupled with different architecture) means more cells to work at the same time, so can increase the processing speed several times. In reality processing speed is also limited by data transfer between disk, memory and processor cores, but this is not the main limitation. There are several solutions for these kinds of tasks. The classical solution is distributed computing (Horváth, 2005); the now more-emerging solution is the possibility of using the graphical processing units (GPUs) (Rouhipour, Bentley, & Shayani, 2010) inside a single computer, or a combination of both.

If our framework used distributed computing, it would mean that one VLO is spanned across many computers. The limiting factor here is the link between computers which is much slower than communications within a single computer. So the main task would be to limit the need for network communication and share the VLO so each has a part that communicates internally most of the time. Also, sending information at the moment relies heavily on the local

memory, not all the data is sent from one cell to the other, but only a pointer that points to the data in memory. This makes communications much faster as it does not need big chunks of data to be duplicated. However a pointer cannot travel across a network and still be valid. By design the blood cells are responsible for any communications, so they can easily be modified to detect if a destination is on another computer. In this case, they need to pick up the physical data and transfer it to the receiving computer, or any equivalent mechanism, e.g., the real data is only sent to the real destination and not through all the computers in between the sender and the destination where only a special pointer would travel. This requires that the data can be serialised, which means that it can be converted into a data stream. Each cell would be required in this case to provide a job serialisation function much like they are required to provide some functions for the optimisation subsystem if they want to use it. With these serialisation functions the blood cells could convert the data to a data stream when leaving the computer and take that data with them.

Every computer would have an entry point and an exit point from a networking point of view. The entry point would normally be an organ with a main tissue that handles networking, and the exit point would be the last organ's last tissue on that particular computer. The entry point would have a cell that monitors network traffic and if it detects an incoming blood cell, it creates a thread for it (with priority, meaning that it might need to ignore the current cell count) and gives it the data it was sent with. Then the blood cell continues its work and looks for the destination on the new computer. If it finds it, the data is converted back with the deserialization function of the cell that's job the data holds. If the destination is not on this computer, then the blood cell reaches the last tissue on the local computer and sees that it needs to jump again into another computer. This last tissue can either be a standard one, and in this case the blood cell needs to use the network interface directly and send itself across, after which it destroys itself in the local computer, or it would be safer and better if the last tissue in each computer would be a special support tissue that handles the network sending and the destruction of the blood cell (Figure 19).

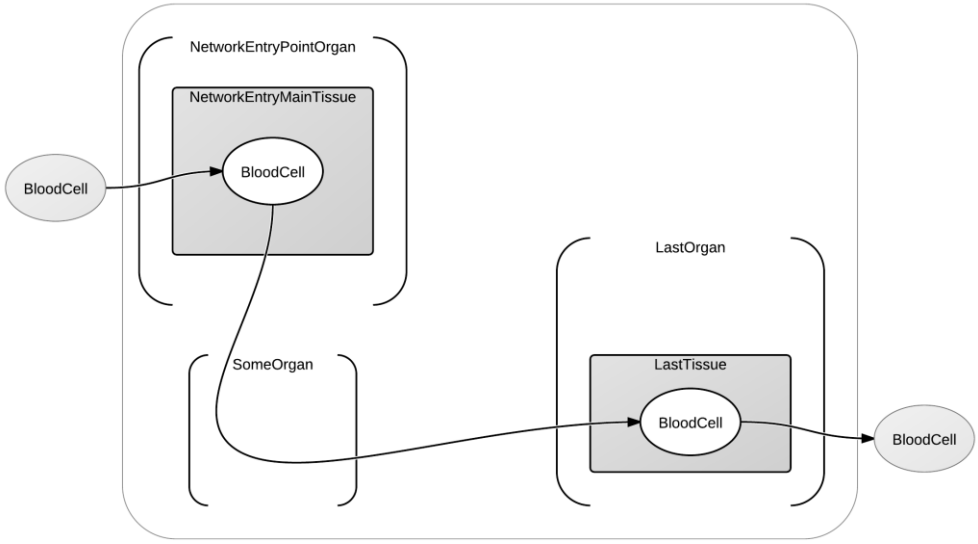


Figure 19 – Distributed virtual living organism on one computer.

The blood cell coming from the network is physically recreated by a special organ's main tissue. If the blood cell does not find the address on the local computer, then a special support tissue that is the last tissue on the computer sends it to the next node via the local network.

With this modified structure and communications network, the new VLO would be able to span across many computers. But it would be unwise to just do an arbitrary division of the organism as the goal is to minimise network traffic. Luckily most tissues are created with the intent at having all the necessary cells they need for their primary function. An initial division could be to have one tissue per processor, but it would hardly be optimal. A tissue uses the services of the support tissues in its organ, so if it is not too much for one computer, a single organ could be put on each computer.

Of course it could easily happen that a group of organs have functions that require great processing power, while other organs are idle most of the time and

do not require a separate computer. At this stage, the distribution of the organs can be manual, in which case the user would decide what is the optimal distribution of organs and tissue across the network, or there could be an automatic subsystem that first analyses a test run of the VLO and creates an optimal distribution pattern according to the results.

Blood cells could log when they leave a particular unit, be it a tissue or an organ, and the resulting data would create a graph that can be processed with several graph algorithms that can find closely coupled groups with a maximum group number below the number of computers on the network and of roughly equal processing requirements. There are several exising algorithms that can help to distribute the workload in a distributed system (Hendricksona & Kolda, 2000). In some cases if a single tissue requires most of the processing power inside a VLO, it might be possible to create that tissue across several computers.

Since jobs target cell types and not cells, it is possible to have the same tissue with the same type of cells on different computers, and then the only problem is to distribute the number of incoming jobs between them. In that case they need to predict and routinely synchronise the job count of their cell types and then the incoming job can be sent to the one with the minimal amount of jobs waiting in its queue for the target cell type. In this case jobs that have an address in the local computer would have priority locally even if another copy has fewer jobs for the same type of cell, because network traffic must be avoided whenever possible. With these not too difficult modifications (and some minor others), an effective distributed version of the framework could be created. Of course every VLO previously created would need to be modified so every cell of it provides the serialisation function.

And aside of the previously mentioned modifications, some internal addressing methods would need to change as in some instances memory addresses are used, and this would need to be revised not to cause any trouble.

The other method of speeding up the use of the framework would be to use the power of GPUs where possible. These methods have some structural limitations as these units are designed to be used with special functionalities in graphical processing, so not all the functions could be made faster with these, but as there can be several hundred of these in one computer. This technique could be used together with distributed computing for an even greater performance boost.

10 Conclusions: Main contributions to knowledge

The main contribution of this research to knowledge is the idea of emulating biology in general, and the methods to achieve this. The programming paradigm described in this thesis not only shows how to achieve goals using the VLO framework, but is designed to be a base of any future endeavours to create similar systems in other environments if needed. The included examples give a good view of what the framework is capable of and show the path to create similar biological emulations. The virtual cancer demonstration is the first instance of cancer that has ever been emulated as a result of spontaneously arising aneuploidy, and not only shows the benefits of this level of complexity and abstraction, but even gives some insight into how a malignant tumour can emerge from a benign one, namely by changes in the control mechanism of the cell and independence of location the cell resides in. Result data also shows what the effects of leaving out of multiplying each individual chromosome of the test system can be. This demonstration also hinted at ways the framework could be used by the drug industry by finding critical levels of aneuploidy as an indicator of malignant cancer. The many suggested examples of future use and extensions with plans of executions and discussion of possibilities make it easy to see the direction and possibilities of this method and makes it easier to be used for those who seek long-term solutions to currently unsolved problems.

REFERENCES

Models that take drugs. (2005, June). *The Economist*, S23–S24.

littleb.org - home of the little b modular modeling language. (2008). Retrieved 03 10, 2011, from littleb.org: http://www.littleb.org/

artificiallife.org. (2010). Retrieved 03 10, 2011, from artificiallife.org: http://www.artificiallife.org/

UML. (2010). Retrieved 12 10, 2010, from IBM Corporation Web site: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

Abbas, A., & Lichtman, A. (2000). *Cellular and Molecular Immunology.* Saunders.

Alarcon, T., Byrne, H. M., & Maini, P. K. (2003). A cellular automaton model for tumour growth in inhomogeneous environment. *J Theoret Biol, 225*, 257-274.

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., & Walter, P. (2002). *Molecular Biology of the Cell* (4th ed.). New York: Garland.

Allan, G., Yang, R., Fotheringham, A., & Mather, R. (2001). Neural modelling of polypropylene fiber processing: predicting the structure and properties and identifying the control parameters for specified fibers. *Journal of Materials Science, 36*, 3113-3118.

An, G. (2001). Agent-Based Computer Simulation and Sirs: Building A Bridge Between Basic Science and Clinical Trials. *Shock, 16*(4), 266-273.

An, G. (2004). In silico experiments of existing and hypothetical cytokine-directed clinical trials using agent-based modeling. *Critical Care Medicine, 32*(10), 2050-2060.

An, G. (2008). Introduction of an agent-based multi-scale modular architecture for dynamic knowledge representation of acute inflammation. *Theor Biol Med Model*, 5:11.

Antal, M., Böde, C., & Csermely, P. (2009). Perturbation waves in proteins and protein networks: Applications of percolation and game theories in signaling and drug design. *Curr. Prot. Pept. Sci., 10*, 161-172.

Arnold, D. V., & Beyer, H.-G. (2006). Optimum tracking with evolution strategies. *Evolutionary Computation, 14*(3), 291–308.

Bäck, T., Fogel, D. B., & Michalewicz, Z. (2000). *Evolutionary computation 1 basic algorithms and operators.* Bristol and Philadelphia: Institute of Physics Publishing.

Bäck, T., Fogel, D. B., & Michalewicz, Z. (2000). *Evolutionary computation 2 advanced algorithms and operators.* Bristol and Philadelphia: Institute of Physics Publishing.

Bailey, A., Thorne, B., & Peirce, S. (2007). Multi-cell agent-based simulation of the microvasculature to study the dynamics of circulating inflammatory cell trafficking. *J Biomed Eng, 35*, 916-936.

Balmann, A. (1997). Farm based modelling of regional structural change. A cellular automata approach. *Eur. Rev. Agric. Econ., 24*, 85-108.

Bándi, G., & Ramsden, J. J. (2010). Biological programming. *Journal of Biological Physics and Chemistry, 10*(1), 39–43.

Bassingthwaighte, J. (1995). Toward modelling the human physionome. In S. Sideman, & R. Beyar (Eds.), *Molecular and Subcellular Cardiology: Effects on Structure and Function* (pp. 445–461). New York: Plenum Press.

Bassingthwaighte, J., Chizeck, H., Atlas, L., & Qian, H. (2005). Multi-scale modeling of cardiac energetics. *Ann. NY Acad. Sci, 1047*, 395–426.

Beni, G. (1988). The concept of cellular robotic systems. *Proc of the IEEE Int. Symp. on Intelligent Control*, (pp. 57–62).

Beni, G., & Wang, J. (1989). Swarm intelligence. *Proc of the 7th annual meeting of the robotics society of Japan*, (pp. 425–428).

Bentley, K., Gerhardt, H., & Bates, P. A. (2007). Agent-based simulation of notch-mediated tip cell selection in angiogenic sprout initialisation. *JTheor Biol, 250*, 25-36.

Bentley, P. J. (1999). *Evolutionary design by computers.* Morgan Kaufmann.

Bentley, P. J., & Corne, D. W. (2001). *Creative evolutionary systems.* Morgan Kaufmann.

Bersini, H., & Varela, F. (1991). Hints for adaptive problem solving gleaned from immune network. In H. Schwefel, & H. Mühlenbein (Eds.), *Parallel problem solving from nature* (pp. 343–354). Springer-Verlag.

Beyer, H.-G. (2001). *Theory of evolution strategies.* Springer.

Bilchev, G., & Parmee, I. C. (1995). The ant colony metaphor for searching continuous design spaces. *Evolutionary Computing, AISB workshop*, (pp. 25-39).

Bishop, C. M. (1996). *Neural networks for pattern recognition.* Oxford University Press.

Blackwell, T. M. (2003). Swarms in dynamic environments. *Proceedings of the genetic and evolutionary computation conference 2003 (GECCO 2003), Swarms in dynamic environments*, *2723*, pp. 1-12. Chicago, IL, USA.

Blum, C., & Dorigo, M. (2004). The hyper-cube framework for ant colony optimization. *IEEE Trans Syst Man Cybernet Part B, 34*(2), 1161-1172.

Bonabeau, E., & Théraulaz, G. (2000, March). Swarm smarts. *Scientific American*, 72-79.

Bonabeau, E., Dorigo, M., & Théraulaz, G. (1999). *Swarm intelligence: from natural to artificial systems.* Oxford University Press.

Bonabeau, E., Dorigo, M., & Théraulaz, G. (2000). Inspiration for optimization from social insect behavior. *Nature, 406*, 39-42.

Bousquet, F., & Page, C. L. (2004). Multi-agent simulations and ecosystem management: a review. *Ecological Modelling, 176*, 313-332.

Bousquet, F., Barreteau, O., Page, C. L., Mullon, C., & Weber, J. (1999). An environmental modelling approach: the use of multi-agent simulations. In F. Blasco (Ed.), *Advances in Environmental and Ecological Modelling.* Editions Scientifiques Et.

Bousquet, F., Cambier, C., Mullon, C., Morand, P., Quensiere, J., & Pave, A. (1993). Simulating the interaction between a society and a renewable resource. *Journal of biological systems, 1*, 199-214.

Bradley, D., & Tyrrell, A. (2000). Immunotronics: Hardware fault tolerance inspired by the immune system. *Proceedings of the 3rd International conference on Evoluable Systems (ICES2000)* (pp. 11-20). Springer-Verlag.

Brits, R., Engelbrecht, A. P., & van den Bergh, F. (2003). Scalability of niche PSO. *Proceedings of the IEEE swarm intelligence symposium 2003 (SIS 2003)*, (pp. 228-134). Indianapolis, Indiana, USA.

Bugrim, A., Nikolskaya, T., & Nikolsky, Y. (2004). Early prediction of drug metabolism and toxicity: systems biology approach and modeling. *Drug Discov. Today, 9*, 127-135.

Burrage, K., Tian, T., & Burrage, P. (2004). A multi-scaled approach for simulating chemical reaction systems. *Prog. Biophys. Mol. Biol., 85*, 217–234.

Campbell, K., Razumova, M., Kirkpatrick, R., & Slinker, B. (2001). Nonlinear myofilament regulatory processes affect frequency dependent muscle fiber stiffness. *Biophys. J., 81*, 2278–2296.

Cao, Y., & Dasgupta, D. (2003). An Immunogenetic Approach in Chemical Spectrum Recognition. In Ghosh, & Tsutsui (Eds.), *Advances in Evolutionary Computing.* Springer-Verlag.

Cazangi, R. R., Von Zuben, F. J., & Figueiredo, M. F. (2006). Stigmergic autonomous navigation in collective robotics. In A. Abraham, C. Grosan, & V. Ramos (Eds.), *Stigmergic optimization* (pp. 25-64). Berlin: Springer.

Chen, M., & Hofestadt, R. (2006). A medical bioinformatics approach for metabolic disorders: biomedical data prediction, modeling, and systematic analysis. *J. Biomed. Inform., 39*, 147-159.

Cho, C. R., Labow, M., Reinhardt, M., Oostrum, J. v., & Peitsch, M. C. (2006). The application of systems biology to drug discovery. *Curr. Opin. Chem. Biol., 10*, 294–302.

Christiansen, M., & Kirby, S. (2003). Language and evolution: consensus and controversies. *Trends in Cognitive Science, 7*(7), 300-307.

Coath, G., & Halgamuge, S. K. (2003). A comparison of constraint-handling methods for the application of particle swarm optimization to constrained nonlinear optimization problems. *Proceedings of IEEE congress on evolutionary computation 2003 (CEC 2003)*, (pp. 2419–2425). Canbella, Australia.

Corne, D. W., & Bentley, P. J. (2001). *Creative evolutionary systems.* Morgan Kaufmann.

Crampin, E. J., Halstead, M., Hunter, P., Nielsen, P., Noble, D., Smith, N., & Tawhai, M. (2004). Computational physiology and the physiome project. *Exp. Physiol., 89*, 1-26.

Cutello, V., & Nicosia, G. (2004). The clonal selection principle for in silico and in vitro computing. In L. N. de Castro, & F. J. Von Zuben (Eds.), *Recent developments in biologically inspired computing* (pp. 104-146). Idea Group Publishing.

Czernichow, T., Germond, A., Dorizzi, B., & Caire, P. (1995). Improving recurrent network load forecasting. *Proc of the IEEE Int Conf ICNN '95*, (pp. 899-904). Perth.

Dallon, J., Jang, W., & Gomer, R. H. (2006). Mathematically modelling the effects of counting factor in Dictyostelium discoideum. *MathMed Biol, 23*, 45-62.

Dasgupta, D. (1996). Using Immunological Principles in Anomaly Detection. *Proc. of the Artificial Neural Networks in Engineering (ANNIE'96).* St. Louis.

Dasgupta, D. (1997). Artificial Neural Networks and Artificial Immune Systems: Similarities and Differences. *1997 IEEE International Conference on Systems Man and Cybernetics Computational Cybernetics and Simulation. 1*, pp. 873-878. IEEE.

Dasgupta, D. (2006). Advances in artificial immune systems. *IEEE Computational Intelligence Magazine, 1*(4), 40-49.

Dasgupta, D., & Forrest, S. (1996). Novelty Detection in Time Series Data using Ideas from Immunology. *ISCA 5th International Conference on Intelligent Systems.* Reno.

Dasgupta, D., & Michalewicz, Z. (1997). *Evolutionary algorithms in engineering applications.* Springer.

Dasgupta, D., KrishnaKumar, K., Wong, D., & Berry, M. (2004). Negative Selection Algorithm for Aircraft Fault Detection. *Proceedings of the Third International Conference, ICARIS 2004 on Artificial Immune Systems.* Catania.

de Castro, L. N. (2005). Natural computing. In M. Khosrow-Pour (Ed.), *Encyclopedia of information science and technology* (Vol. IV, pp. 2080–2084). Idea Group Inc.

de Castro, L. N. (2006). *Fundamentals of natural computing: basic concepts, algorithms, and applications.* CRC Press LLC.

de Castro, L. N. (2007). Fundamentals of natural computing: an overview. *Physics of Life Reviews, 4*, 1-36.

De Castro, L. N., & Von Zuben, F. J. (2000). An evolutionary immune network for data clustering. In F. M. Franca, & C. H. Ribeiro (Ed.), *Proceedings of 6th Brazilian Symposium on Neural Networks (SBRN 2000)* (pp. 84–89). IEEE Computer Society.

de Castro, L. N., & Von Zuben, F. J. (2001). aiNet: an artificial immune network for data analysis. In H. A. Abbas, R. A. Sarker, & C. S. Newton (Eds.), *Data mining: a heuristic approach* (pp. 231-259). Idea Group Publishing.

de Castro, L. N., & Von Zuben, F. J. (2002). Learning and optimization using the clonal selection principle. *IEEE Transactions on Evolutionary Computation, 6*(3), 239-251.

de Castro, L. N., & Von Zuben, F. J. (2004). *Recent developments in biologically inspired computing.* Idea Group Publishing.

de Jong, H. (2002). Modeling and simulation of genetic regulatory systems: a literature review. *J. Comput. Biol., 9*, 67-103.

Deadman, P., & Gimblett, H. (1994). A role for goal-oriented autonomous agents in modeling people–environment interactions in forest recreation. *Math. Comput. Model., 20*, 121-133.

Dean, J., Gumerman, G., Epstein, J., Axtell, R., Swedlund, A., . . . McCaroll, S. (2000). Understanding anasazi culture change through agent-based

modeling. In T. Kohler, & G. Gumerman (Eds.), *Dynamics in Human and Primate Societies* (pp. 179-206). Oxford University Press.

Deneubourg, J., & Goss, S. (1989). Collective patterns and decision making. *Ethol. Ecol. Evol., 1*, 295-311.

Derry, J. (1998). Modelling ecological interaction despite object-oriented modularity. *Ecol. Model., 107*, 145-158.

Deutschman, D., Levin, S., Devine, C., & Buttel, L. (1997). Scaling from trees to forests: analysis of a complex simulation model. *Sience Online*. Retrieved 07 29, 2011, from http://www.sciencemag.org/site/feature/data/deutschman/index.htm

Dorigo, M., & Di Caro, G. (1999). The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, & G. F (Eds.), *New ideas in optimization* (pp. 13-49). McGraw-Hill.

Dorigo, M., & Stützle, T. (2004). *Ant colony optimization.* MIT Press.

Dorigo, M., Di Caro, G., & Gambardella, L. M. (1999). Ant algorithms for discrete optimization. *Artificial Life, 5*(3), 137-172.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. *IEEE Trans Syst Man Cybernet Part B, 26*(1), 29-41.

Duesberg, P. (2007). Chromosomal Chaos and Cancer. *Scientific American*(296), 52-59.

Dumont, B., & Hill, D. (2001). Multi-agent simulation of group foraging in sheep: effects of spatial memory, conspecific attraction and plot size. *Ecol. Model., 141*, 201-215.

Emmeche, C. (2000). Closure, function, emergence, semiosis and life: the same idea? Reflections on the concrete and the abstract in theoretical

biology. In J. L. Chandler, & G. Van de Vijver (Eds.), *Closure: emergent organizations and their dynamics. Annals of the New York Academy of Sciences* (pp. 187-197). New York: The New York Academy of Sciences.

Emmeche, C., Koppe, S., & Stjernfelt, F. (1997). Explaining emergence: towards an ontology of levels. *Journal for General Philosophy of Science, 28*, 83-119.

Emonet, T., Macal, C. M., North, M. J., Wickersham, C. E., & Cluzel, P. (2005). AgentCell: a digital single-cell assay for bacterial chemotaxis. *Bioinformatics, 21*(11), 2714-2721.

Engelbrecht, A. P. (2006). *Fundamentals of computational swarm intelligence.* John Wiley & Sons.

Epstein, J. M., & Axtell, R. L. (1996). *Growing Artificial Societies: Social Science from the Bottom Up (Complex Adaptive Systems).* The MIT Press.

Fabarius, A., Li, R., Yerganian, G., Hehlmann, R., & Duesberg, P. (2008). Specific clones of spontaneously evolving karyotypes generate individuality of cancers. *Cancer Genetics and Cytogenetics*(180), 89-99.

Farina, M., Deb, K., & Amato, P. (2004). Dynamic multiobjective optimization problems: test cases, approximations, and applications. *IEEE Trans Evolutionary Computation, 8*(5), 425-442.

Farkas, I., Korcsmáros, T., Kovács, I., Mihalik, Á., Palotai, R., Simkó, G., . . . Csermely, P. (2011). Network-based tools in the identification of novel drug-targets. *Science Signaling, 4*(173).

Farmer, J. D., & Belin, A. (1991). Artificial life: the coming evolution. In C. Langton, C. Taylor, J. D. Farmer, & S. Rasmussen (Eds.), *Artificial life II* (pp. 815-840).

Farmer, J. D., Packard, N. H., & Perelson, A. S. (1986). The immune system, adaptation, and machine learning. *Physica D, 22*, 187-204.

Farnsworth, M., Benkhelifa, E., Tiwari, A., & Zhu, M. (2010). A Multi-level Evolutionary Design Optimisation of MEMS: Methodology and Application. *Proceedings Third International Conference on Multidisciplinary Design Optimization and Applications.*

Farnsworth, M., Benkhelifa, E., Tiwari, A., & Zhu, M. (2010). A Novel Approach to Multi-level Evolutionary Design Optimization of a MEMS Device. *Lecture Notes in Computer Science, 6274/2010*, 322-334.

Fausett, L. (1994). *Fundamentals of neural networks: architectures, algorithms, and applications.* Prentice-Hall.

Fogel, D. B. (Ed.). (1998). *Evolutionary computation: the fossil record.* The IEEE Press.

Fogel, D. B. (2000). *Evolutionary computation: principles and practice for signal processing.* The International Society for Optical Engineering.

Forrest, S., Javornik, B., Smith, R. E., & Perelson, A. S. (1993). Using genetic algorithms to explore pattern recognition in the immune system. *Evolutionary Computation, 1*(3), 191-211.

Forrest, S., Perelson, A. S., Allen, L., & Cherukuri, R. (1994). Self–nonself discrimination in a computer. *Proceedings of the 1994 IEEE Symposium on Security and Privacy* (p. 202). IEEE Computer Society.

Frank, S. A. (1996). *The Design of Natural and Artificial Adaptive Systems.* New York: Academic Press.

Freitas, A. A., & Rozenberg, G. (2002). *Data mining and knowledge discovery with evolutionary algorithms.* Springer.

Fu, L. (1994). *Neural Networks in Computer Intelligence.* McGraw-Hill.

Galam, S. (2011). Collective beliefs versus individual inflexibility: The unavoidable biases of a public debate. *Physica A, 390*, 3036-3054.

Galam, S., Chopard, B., & Droz, M. (2002). Killer Geometries in Competing Species Dynamics. *Physica A, 314*(1-4), 256-263.

Galeano, J. C., Veloza-Suan, A., & González, F. A. (2005). A comparative analysis of artificial immune network models. *Proc of the genetic and evolutionary computation conference*, (pp. 361-368).

Garny, A., Kohl, P., & Noble, D. (2003). Cellular open resource (COR): a public CellML based environment for modelling biological function. *Int. J. Bif. Chaos, 13*, 3579–3590.

Gergic, J. (2003). Towards a Versioning Model for Component-based Software Assembly. *Proceedings of the International Conference on Software Maintenance (ICSM'03).* Washington DC: IEEE Computer Society.

Ghosh, S., Matsuoka, Y., & Kitano, H. (2010). Connecting the dots: role of standardization and technology sharing in biological simulation. *Drug Discovery Today, 15*(23/24), 1024-1031.

Gilbert, N., & Doran, J. (1994). *Simulating Societies.* UCL Press.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning.* Reading, MA: Addison-Wesley.

Goldman, J., Gullick, W., & Johnson, C. (2004). Individual-based simulation of the clustering behaviour of epidermal growth factor receptors. *Scientific Programming, 12*, 25-43.

González, F., & Dasgupta, D. (2003). Anomaly detection using real-valued negative selection. *Genetic Programming and Evolvable Machines, 4*(4), 383-403.

Hayes, N., Howard-Cofield, E., & Gullick, W. (2004). Green fluorescent protein as a tool to study epidermal growth factor receptor function. *Cancer Lett, 206*(2), 129-35.

Haykin, S. (1999). *Neural networks: a comprehensive foundation* (2nd ed.). Prentice-Hall.

Heiner, M., Koch, I., & Will, J. (2004). Model validation of biological pathways using Petri nets--demonstrated for apoptosis. *Biosystems, 75*, 15-28.

Hendricksona, B., & Kolda, T. G. (2000). Graph partitioning models for parallel computing. *Parallel Computing, 26*(12), 1519-1534.

Higashi, N., & Iba, H. (2003). Particle swarm optimization with Gaussian mutation. *Proceedings of the IEEE swarm intelligence symposium 2003 (SIS 2003)*, (pp. 72-79). Indianapolis, Indiana, USA.

Hightower, R. R., Forrest, S. A., & Perelson, A. S. (1995). The evolution of emergent organization in immune system gene libraries. In L. J. Eshelman (Ed.), *Proc of the 6th Int conference on genetic algorithms* (pp. 344-350). Morgan Kaufmann.

Hoffmann, G. W. (1986). A neural network model based on the analogy with the immune system. *Journal of Theoretical Biology, 122*, 33-67.

Hofmeyr, S. A., & Forrest, S. (2000). Architecture for an artificial immune system. *Evolutionary Computation, 7*(1), 45-68.

Hood, L. (2003). Systems biology: integrating technology, biology, and computation. *Mech. Ageing Dev., 124*, 9–16.

Horváth, Z. (2005). Parallel and distributed programming. (in Hungarian). In Á. Fóthi, & Z. Horváth, *Introduction to programming. (in Hungarian)* (Vol. III).

Hruschka, E. R., & Ebecken, N. (2006). Extracting rules from multilayer perceptrons in classification problems: a clustering-based approach. *Neurocomputing, 70*(1-3), 384-397.

Humphrey, J. (2003). Continuum biomechanics of soft biological tissues. *Proc. R. Soc. A, 459*, 3–46.

Hunt, J. E., & Cooke, D. E. (1996). Learning using an artificial immune system. *Journal of Network and Computer Applications, 19*, 189-212.

Hunter, P., & Borg, T. (2003). Integration from proteins to organs: the Physiome Project. *Nat. Rev., 4*, 237–243.

Hunter, P., Robbins, P., & Noble, D. (2002). The IUPS human physiome project. *Eur. J. Physiol., 445*, 1-9.

Huston, M., DeAngelis, D., & Post, W. (1988). New computer models unify ecological theory. *Bioscience, 38*, 682-691.

Hyde, P. (1999). Thread Programming. In P. Hyde, *Java Thread Programming.* Sams.

Ideker, T., Galitski, T., & Hood, L. (2001). A new approach to decoding life: systems biology. *Annu. Rev. Genomics Hum. Genet., 2*, 343–372.

Jacobsson, H. (2005). Rule extraction from recurrent neural networks: a taxonomy and review. *Neural Computation, 17*(6), 1223-1263.

Janssen, M., & Carpenter, S. (1999). Managing the Resilience of Lakes: A Multi-agent Modeling Approach. *Conservation Ecology, 3*(2), 15.

Jerne, N. (1974). Towards a network theory of the immune system. *Ann. Immunol., 125C*(1-2), 373-389.

Johnson, C. G., Goldman, J. P., & Gullick, W. J. (2004). Simulating complex intracellular processes using object-oriented computational modelling. *Progress in Biophysics & Molecular Biology*(86), 379–406.

Johnson, S. (2002). *Emergence: the connected lives of ants, brains, cities and software.* Penguin Books.

Johnston, T. D., & Turvey, M. T. (1980). A Sketch of an Ecological Metatheory for Theories of Learning. In G. Bower (Ed.), *The psychology of learning and motivation* (Vol. 14, pp. 147-205). New York: Academic Press.

Jung, S., & Hsia, T. C. (1998). Neural network impedance force control of robot manipulator. *IEEE Trans. Ind. Electron., 45*, 451–461.

Kalogirou, S. (1999). Applications of artificial neural networks in energy systems: A review. *Energy Conversion & Management, 40*, 1073-1087.

Kalogirou, S. A., Neocleous, C. C., & Schizas, C. N. (1996). Artificial neural networks in modelling the heat-up response of a solar steam generation plant. *Proc of the Int Conf EANN '96*, (pp. 1-4). London.

Kalogirou, S. A., Neocleous, C. C., & Schizas, C. N. (1997). Artificial neural networks for the estimation of the performance of a parabolic trough collector steam generation system. *Proc of the Int Conf EANN '97*, (pp. 227-232). Stockholm.

Kansal, A. R., Torquato, S., Chiocca, E. A., & Deisboeck, T. S. (2000). Emergence of a Subpopulation in a Computational Model of Tumor Growth. *Journal of Theoretical Biology, 207*, 431-441.

Karr, C. L., & Freeman, L. M. (1998). *Industrial applications of genetic algorithms.* CRC Press.

Keedwell, E., & Narayanan, A. (2003). Genetic algorithms for gene expression analysis. In G. Raidl, S. Cagnoni, J. Cardalda, D. Corne, J. Gottlieb, A. Guillot, . . . M. Middendorf (Ed.), *Applications of Evolutionary Computing: Evoworkshops.* Berlin: Springer.

Keedwell, E., Narayanan, A., & Savic, D. (2002). Constructing gene regulatory networks using artificial neural networks. *Proceedings of the 2002 International Joint Conference on Neural Networks.* Honolulu.

Keeley, B. L. (1997). Evaluating artificial life and artificial organisms. In C. Langton, & K. Shimohara (Eds.), *Artificial life V* (pp. 264-271). Addison-Wesley.

Keener, J., & Sneyd, J. (1998). *Mathematical Physiology.* New York: Springer.

Kelsey, J., & Timmis, J. (2003). Immune inspired somatic contiguous hypermutation for function optimisation. *Proc of the genetic and evolutionary computation conference. Lecture notes in computer science. 2723*, pp. 207-218. Springer.

Kennedy, J. (1997). The particle swarm: social adaptation of knowledge. *Proc of the IEEE Int Conf on evolutionary computation*, (pp. 303-308).

Kennedy, J. (2004). Particle swarms: optimization based on sociocognition. In L. N. de Castro, & F. J. Von Zuben (Eds.), *Recent developments in biologically inspired computing* (pp. 235-269). Idea Group Publishing.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Proc of the IEEE Int Conf on neural networks*, *4*, pp. 1942-1948. Perth, Australia.

Kennedy, J., Eberhart, R., & Shi, Y. (2001). *Swarm intelligence.* Morgan Kaufmann.

Kephart, J. (1994). A biologically inspired immune system for computers. *Proceedingsof the Fourth International Workshop on Synthesis and Simulatoin of Living Systems, Artificial Life IV*, (pp. 130–139).

Kerckhoffs, R., Bovendeerd, P., Kotte, J., Prinzen, F., Smits, K., & Arts, T. (2003). Homogeneity of cardiac contraction despite physiological asynchrony of depolarization: a model study. *Ann. Biomed. Eng., 31*, 536-547.

Kerckhoffs, R., Faris, O., Bovendeerd, P., Prinzen, F., Smits, K., McVeigh, E., & Arts, T. (2005). Electromechanics of paced left ventricle simulated by straightforward mathematical model: comparison with experiments. *Am. J. Physiol. Heart Circ. Phys., 289*, H1889–H1897.

Kerckhoffs, R., Healy, S., Usyk, T., & McCulloch, A. (2006). Computational methods for cardiac electrophysiology. *Proc. IEEE, 94*, 769–783.

Khotanzad, A., Abaye, A., & Maratukulam, D. (1995). An adaptive and modular recurrent neural network based power system load forecaster. *Proc of the IEEE Int Conf ICNN '95*, *2*, pp. 1032-1036. Perth.

Kier, L. B., Cheng, C. K., Testa, B., & Carrupt, P.-A. (1996). A cellular automata model of enzyme kinetics. *J. Mol. Graph., 14*, 227–231.

Kier, L. B., Cheng, C.-K., Testa, B., & Carrupt, P.-A. (1996). A cellular automata model of micelle formation. *Pharm. Res., 13*, 1419–1422.

Kier, L. B., Cheng, C.-K., Testa, B., & Carrupt, P.-A. (1997). A cellular automata model of diffusion in aqueous systems. *J. Pharm. Sci., 86*, 774–778.

Kim, J., Bentley, P. J., Aickelin, U., Greensmith, J., Tedesco, G., & Twycross, J. (2007). Immune system approaches to intrusion detection – a review. *Natural Computing, 6*, 413-466.

Kirby, S. (2002). Natural language from artificial life. *Artif Life, VIII*, 185-215.

Kitano, H. (2002). Computational systems biology. *Nature, 420*, 206-210.

Kohl, P., Noble, D., Winslow, R., & Hunter, P. (2000). Computational modelling of biological systems: tools and visions. *Philos. Trans. R. Soc. London A, 358*, 579–610.

Kohler, T., & Carr, E. (1996). *Swarm Based Modelling of Prehistoric Settlement Systems in Southwestern North America.* Forli, Italy.

Kohonen, T. (2000). *Self-organizing maps.* Springer.

Kondoh, M. (2003). High reproductive rates result in high predation risks: a mechanism promoting the coexistence of competing prey in spatially structured populations. *Am. Nat., 161*, 299-309.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection.* MIT Press.

Koza, J. R. (1994). *Genetic programming II: automatic discovery of reusable programs.* MIT Press.

Krishnakumar, K., & Neidhoefer, J. (1999). Immunized Adaptive Critic for an Autonomous Aircraft Control Application. In *Artificial Immune Systems and Their Applications* (pp. 221-240). Springer-Verlag.

Kube, C. R., Parker, C. A., Wang, T., & Zhang, H. (2004). Biologically inspired collective robotics. In L. N. de Castro, & F. J. Von Zuben (Eds.), *Recent developments in biologically inspired computing* (pp. 367-397). Idea Group Publishing.

Kumar, N., Hendriks, B. S., Janes, K. A., Graaf, D. d., & Lauffenburger, D. A. (2006). Applying computational modeling to drug discovery and development. *Drug Discov. Today, 11*, 806-811.

Kushchu, I. (2005). Web-based evolutionary and adaptive information retrieval. *IEEE Trans Evolutionary Computation, 9*(2), 117-125.

Lamb, T. (1996). Stochastic simulation of activation in the G-protein cascade. *Biophys., 67*, 1439–1454.

Langton, C. G. (1988). Artificial life. In C. G. Langton (Ed.), *Artificial life* (pp. 1-47). Addison-Wesley.

Langton, C. G. (1990). Computation at the edge of Chaos: Phase-Transitions and Emergent Computation. *Physica D, 42*, 12-37.

Lansing, J., & Kremer, J. (1994). Emergent Properties of Balinese Water Temple Networks: Coadaptaion on a Rugged Fitness Landscape. In L. C. (Ed.), *Artificial Life III.* Santa Fe: Addison-Wesley.

Le Boudec, J., & Sarafijanovic, S. (2003). *An artificial immune system approach to misbehavior detection in mobile ad-hoc networks.* Technical Report, Ecole Polytechnique Federale de Lausanne.

Le Boudec, J., & Sarafijanovic, S. (2004). An artificial immune system approach to misbehavior detection in mobile ad-hoc networks. *Proceedings of Bio-ADIT 2004 (The First International Workshop on Biologically Inspired Approaches to Advanced Information Technology)*, (pp. 96–111). Lausanne.

Lees, M., Logan, B., & King, J. (2007). Multiscale models of bacterial populations. *Proc 2007 Winter Simulat Conf*, (pp. 860–869).

LeGrice, I., Smaill, B., Chai, L., Edgar, S., Gavin, J., & Hunter, P. (1995). Laminar structure of the heart: ventricular myocyte arrangement and connective tissue architecture in the dog. *Am. J. Physiol., 269*, H571–H582.

Lindgren, K., & Nordahl, M. (1994). Artificial food webs. In C. Langton (Ed.), *Artificial Life III.* Addison-Wesley.

Lloyd, C., Halstead, M., & Nielsen, P. (2004). CellML: its future, present and past. *Prog. Biophys. Mol. Biol., 85*, 433–450.

Loew, L., & Schaff, J. (2001, October). The Virtual Cell: A Software Environment for Computational Cell Biology. *Trends in Biotechnology, 19*(10), 401-406.

Lollini, P.-L., Motta, S., & Pappalardo, F. (2006). Discovery of cancer vaccination protocols with a genetic algorithm driving an agent based simulator. *BMC Bioinformatics, 7:352*.

Luthi, P. O., Chopard, B., Preiss, A., & Ramsden, J. J. (1998). A cellular automaton model for neurogenesis in Drosophila. *Physica D, 118*, 15 l-160.

Maley, C. C. (1999). Four steps toward open-ended evolution. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, & R. E. Smith (Ed.), *Proceedings of the genetic and evolutionary computation conference* (pp. 1336-1343). Morgan Kaufmann.

Mallet, D. a. (2006). A cellular automata model of tumor–immune system interactions. *J. Theor. Biol., 239*(3), 334–350.

Mange, D., Sipper, M., Stauffer, A., & Tempesti, G. (2000). Towards robust integrated circuits: The Embryonics approach. *Proceedings of the IEEE, 88*(4), 516–541.

Maniatty, W., Szymanski, B., & Caraco, T. (1993). Implementation and Performance of Parallel Ecological Simulations. *Proceedings IFIP WG10.3 International Conference on Applications in Parallel and Distributed Computing.* Amsterdam: North-Holland Publishing.

Margulis, L., Sagan, D., & Eldredge, N. (2000). *What is life?* University of California Press.

Materi, W., & Wishart, D. S. (2007). Computational systems biology in drug discovery and development: methods and applications. *Drug Discovery Today, 12*(7/8), 295-303.

Matzinger, P. (1994). Tolerance, danger, and the extended family. *Annual Review of Immunology, 12*, 991–1045.

May, R. (1973). *Stability and Complexity in Model Ecosystems.* Princeton University Press.

McCauley, E., Wilson, W., & de Roos, A. (1993). Dynamics of age-structured and spatially structured predator–prey interactions: individual-based models and population-level formulation. *Am. Natur., 142*, 412-442.

McCulloch, W., & Pitts, W. H. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics, 5*, 115-133.

Meireles, M. R., Almeida, P. E., & Simoes, M. G. (2003). A comprehensive review for industrial applicability of artificial neural networks. *Industrial Electronics, IEEE Transactions on, 50*(3), 585-601.

Melnikov, Y., & Tarakanov, A. O. (2003). Immunocomputing model of intrusion detection. *Computer Network Security, Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2003*, (pp. 453-456). St. Petersburg.

Mitchell, M. (1996). *An introduction to genetic algorithms.* MIT Press.

Mitchell, M., Crutchfield, J. P., & Das, R. (1996). Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work. *In Proceedings of the First International Conference on Evolutionary Computation and its Applications (EvCA '96).* Moscow.

Morton-Firth, C., & Bray, D. (1998). Predicting temporal fluctuations in an intracellular signalling pathway. *J. Theor. Biol., 192*, 117–128.

Munakata, T. (1998). *Fundamentals of the New Artificial Intelligence—Beyond Traditional Paradigms.* Berlin: Springer-Verlag.

Naka, S., Genji, T., Yura, T., & Fukuyama, Y. (2003). A hybrid particle swarm optimization for distribution state estimation. *IEEE Trans on Power Systems, 18*(1), 60-68.

Narayanan, A., Keedwell, E., & Olsson, B. (2002). Artificial intelligence techniques for bioinformatics. *Appl. Bioinf., 1*(4), 191-222.

Neal, M. (2003). Meta-stable memory in an artificial immune network. In J. Timmis, P. Bentley, & E. Hart (Ed.), *Proc of the 2nd international conference on artificial immune systems. Lecture notes in computer science. 2787*, pp. 168-180. Springer.

Negroni, J., & Lascano, E. (1996). A cardiac muscle model relating sarcomere dynamics to calcium kinetics. *J. Mol. Cell. Cardiol., 28*, 915–929.

Nickerson, D., Smith, N., & Hunter, P. (2001). A model of cardiac cellular electromechanics. *Philos. Trans. R. Soc. London A, 359*, 1159–1172.

Nickerson, D., Smith, N., & Hunter, P. (2005). New developments in a strongly coupled cardiac electromechanical model. *Europace, 7*, S118–S127.

Nielsen, P., LeGrice, I., Smaill, B., & Hunter, P. (1991). Mathematical-model of geometry and fibrous structure of the heart. *Am. J. Physiol., 260*, H2466–H2476.

Noble, D. (2002). Modeling the heart—from genes to cells to the whole organ. *Science, 295*, 1678–1682.

Noble, D. (2006). Systems biology and the heart. *Biosystems, 83*, 75-80.

Noble, D., & Rudy, Y. (2001). Models of cardiac ventricular action potentials: iterative interaction between experiment and simulation. *Philos. Trans. R. Soc. London A, 359*, 1127–1142.

Nolfi, S., & Floreano, D. (2000). *Evolutionary robotics: the biology, intelligence, and technology of self-organizing machines.* MIT Press/Bradford Books.

Nolfi, S., & Floreano, D. (2002). Synthesis of autonomous robots through evolution. *Trends in Cognitive Science, 6*, 31-37.

Nussey, S., & Whitehead, S. (2001). *Endocrinology - An Integrated Approach.* Oxford: BIOS Scientific Publishers.

O'Neill, B. (2003). Digital evolution. *PLoS Biol, 1*(1), e8.

O'Neill, M., & Ryan, C. (2003). *Grammatical evolution: evolutionary automatic programming in an arbitrary language.* Kluwer Academic.

Ofria, C., & Wilke, C. O. (2004). Avida: a software platform for research in computational evolutionary biology. *Artificial Life, X*, 191-229.

Oprea, M., & Forrest, S. (1998). Simulated evolution of antibody gene libraries under pathogen selection. *Proc of the IEEE System, Man, and Cybernetics*, *4*, pp. 3793 - 3798.

Panait, L., & Luke, S. (2005). Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems, 11*(3), 387-434.

Pang, W., Wang, K.-P., Zhou, C.-G., & Dong, L.-J. (2004). Fuzzy discrete particle swarm optimization for solving traveling salesman problem. *Proc of the fourth international conference on computer and information technology*, (pp. 796–800).

Payeur, P., Le-Huy, H., & Gosselin, C. M. (1995). Trajectory prediction for moving objects using artificial neural networks. *IEEE Trans. Ind. Electron., 42*, 147–158.

Peer, M. A., Shah, N. A., & Khan, K. A. (2004). Cellular automata and its advances to drug therapy for HIV infection. *Indian J. Exp. Biol., 42*, 131–137.

Perelson, A. S., Hightower, R., & Forrest, S. (1996). Evolution and somatic learning in V-region genes. *Research in Immunology, 147*, 202-208.

Perlovsky, L. (2006). Toward physics of the mind: concepts, emotions, consciousness, and symbols. *Physics of Life Reviews, 3*, 23–55.

Pinney, J. W., Westhead, D. R., & McConkey, G. A. (2003). Petri net representations in systems biology. *Biochem. Soc. Trans., 31*, 1513-1515.

Poli, R., Chio, C. D., & Langdon, W. B. (2005). Exploring extended particle swarms: a genetic programming approach. *Proceedings of the 2005 conference on genetic and evolutionary computation*, (pp. 169-176).

Preziosi, L. (Ed.). (2003). *Cancer Modelling and Simulation.* Boca Raton: CRC Press.

Rajesh, S., & Sinha, S. (2008). Measuring collective behaviour of multicellular ensembles: role of space-time scales. *J Biosci, 33*, 289-301.

Ramis-Conde, I., Drasdo, D., Anderson, A. R., & Chaplain, M. A. (2008). Modeling the influence of the E-cadherin-beta-catenin pathway in cancer cell invasion: a multiscale approach. *Biophys J, 95*, 155-165.

Ray, T. S. (1994). An evolutionary approach to synthetic biology. *Artificial Life, 1*(1/2), 179-209.

Ray, T. S., & Hart, J. (1998). Evolution of differentiated multi-threaded digital organisms. *Artificial Life, VI*, 295-304.

Rennard, J.-P. (2004). Perspectives for strong artificial life. In L. N. de Castro, & F. J. Von Zuben (Eds.), *Recent developments in biologically inspired computing* (pp. 301-318). Idea Group Inc.

Resnick, M. (1994). *Turtles, termites, and traffic jams: explorations in massively parallel microworlds.* MIT Press.

Reynolds, C. W. (1987). Flocks, herds, and schools: a distributed behavioral model. *Computer Graphics, 21*(4), 25-34.

Reynolds, C. W. (1999). Steering behaviors for autonomous characters. *Proc of the game developers conference.* [on-line] http://www.red3d.com/cwr/papers/1999/gdc99steer.pdf.

Ridgway, D., Broderick, G., & Ellison, M. J. (2006). Accommodating space, time and randomness in network simulation. *Curr. Opin. Biotechnol., 17*, 493–498.

Riley, S., Fraser, C., Donnelly, C., Ghani, A., Abu-Raddad, L., Hedley, A., . . . Anderson, R. (2003). Transmission dynamics of the etiological agent of SARS in Hong Kong: impact of public health interventions. *Science, 300*, 1961–1966.

Robertson, S. H., Smith, C. K., Langhans, A. L., McLinden, S. E., Oberhardt, M. A., Jakab, K. R., . . . Peirce, S. M. (2007). Multiscale computational

analysis of Xenopus laevis morphogenesis reveals key insights of systems-level behavior. *BMC Systems Biology, 1:46*.

Roese, H., Risenhoover, K., & Folse, J. (1991). Habitat heterogenity and foraging efficiency: an individual-based model. *Ecol. Model., 57*, 133-143.

Room, P., Hanan, J., & Prusinkiewicz, P. (1996). Virtual plants: new perspectives for ecologists, pathologists and agricultural scientists. *Trends in Plant Science, 1*(1), 33-38.

Rouhipour, M., Bentley, P. J., & Shayani, H. (2010). Systemic Computation using Graphics Processors. *Proceedings of the 9th international conference on Evolvable systems: from biology to hardware (ICES'10).* Berlin: Springer-Verlag.

Rowe, G. W. (1994). *The Theoretical Models in Biology.* Oxford University Press.

Ruddon, R. W. (2007). *Cancer biology* (4th ed.). Oxford University Press.

Rudy, Y., & Silva, J. (2006). Computational biology in the study of cardiac ion channels and cell electrophysiology. *Q. Rev. Biophys, 39*, 57–116.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representation by error propagation. In *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1). Cambridge: MIT Press.

Rumelhart, D. E., Widrow, B., & Lehr, M. A. (1994). The basic ideas in neural networks. *Commun. ACM, 37*(3), 87-92.

Sackmann, A., Heiner, M., & Koch, I. (2006). Application of Petri net based analysis techniques to signal transduction pathways. *BMC Bioinformatics, 7:482*.

Saito, Y., Hashimoto, K., Sakurada, T., Fujita, Y., Takahashi, K., & Tomita, M. (2001). Design and development of software environment for whole-cell simulation. *Genome Informatics, 12*, 316–317.

Sanford, C., Yip, M. L., White, C., & Parkinson, J. (2006). Cell++ – simulating biochemical pathways. *Bioinformatics, 22*(23), 2918–2925.

Sarafijanovic, S., & Le Boudec, J. (2003). *An artificial immune system approach with secondary response for misbehavior detection in mobile ad-hoc networks.* Technical Report, Ecole Polytechnique Federale de Lausanne.

Schilstra, M. J., Li, L., Matthews, J., Finney, A., Hucka, M., & Novère, N. L. (2006). CellML2SBML: conversion of CellML into SBML. *Bioinformatics, 22*, 1018-1020.

Schnell, S., Grima, R., & Maini, P. (2007). Multi-scale modeling in biology. *Am. Sci., 95*, 134–142.

Schrödinger, E. (1992). *What is life?: With mind and matter and autobiographical sketches.* Cambridge University Press (First published in 1944).

Schwefel, H.-P. (1965). *Kybernetische Evolution als Strategie der Experimentellen Forschung in der Strömungstechnik.* Berlin: Diploma thesis, Technical University of Berlin.

Shorten, P., & Upreti, G. (2006). A mathematical model of fatty acid metabolism and VLDL assembly in human liver. *Biochim. Biophys. Acta, 1736*, 94-108.

Silvert, W. (1993). Object-oriented ecosystem modelling. *Ecol. Model., 68*, 91-118.

Slepchenko, B. M., Schaff, J. C., Macara, I., & Loew, L. M. (2003). Quantitative cell biology with the Virtual Cell. *Trends Cell Biol., 13*, 570–576.

Smith, N., Nickerson, D., Crampin, E., & Hunter, P. (2004). Multi-scale computational modelling of the heart. *Acta Numer., 13*, 371–431.

Somayaji, A., Hofmeyr, S., & Forrest, S. (1997). Principles of a computer immune system. *Proceeding of New Security Workshop*, (pp. 75–82). Langdale.

Sommerhoff, G. (1950). *Analytical Biology.* London: Oxford University Press.

Southern, J., Pitt-Francis, J., Whiteley, J., Stokeley, D., Kobashi, H., Nobes, R., . . . Gavaghan, D. (2008). Multi-scale computational modelling in biology and physiology. *Progress in Biophysics and Molecular Biology, 96*, 60–89.

Sozinova, O., Jiang, Y., Kaiser, D., & Alber, M. (2005). A three-dimensional model of myxobacterial aggregation by contact-mediated interactions. *Proc. Natl. Acad. Sci. U. S. A., 102*(32), 11308–11312.

Stevens, C., & Hunter, P. (2003). Sarcomere length changes in a 3-D mathematical model of the pig ventricles. *Prog. Biophys. Mol. Biol., 82*, 229–241.

Stoma, S., Lucas, M., Chopard, J., Schaedel, M., Traas, J., & Godin, C. (2008). Flux-based Transport Enhancement as a Plausible Unifying Mechanism for Auxin Transport in Meristem Development. *PLoSComput Biol*, 4:e1000207.

Stroustrup, B. (2000). *The C++ Programming Language (Special Edition).* Addison-Wesley.

Stützle, T., & Hoos, H. H. (2000). MAX–MIN Ant system. *Future Generat Comput Syst, 16*(8), 889-914.

Sveiczer, A., Tyson, J. J., & Novak, B. (2004). Modelling the fission yeast cell cycle. *Brief. Funct. Genomic. Proteomic., 2*, 298–307.

Takahashi, K., Yugi, K., Hashimoto, K., Yamada, Y., Pickett, C., & Tomita, M. (2002, September/October). Computational challenges in cell simulation: a software engineering approach. *IEEE Intelligent Systems*, 64–71.

Timmis, J., Neal, M., & Hunt, J. (2000). An artificial immune system for data analysis. *BioSystems, 55*(1), 143-150.

Timmis, J., Neal, M., & Knight, T. (2002, June). AINE: Machine Learning Inspired by the Immune System. *IEEE Transactions on Evolutionary Computation*.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., . . . Hutchinson, C. (1999). E-CELL: software environment for whole cell simulation. *Bioinformatics, 15*(1), 72–84.

Trapnell, B. C. (2005). A peer-to-peer blacklisting strategy inspired by leukocyteendothelium interaction. In C. Jacob, M. L. Pilat, P. J. Bentley, & J. Timmis (Ed.), *Proceedings of the 4th International Conference on Artificial Immune Systems 2005* (pp. 362–373). Banff: Springer.

Tyson, J. J., & Novak, B. (2001). Regulation of the eukaryotic cell cycle: molecular antagonism, hysteresis, and irreversible transitions. *J Theoret Bio, 210*, 249-263.

Usyk, T., & McCulloch, A. (2003). Relationship between regional shortening and asynchronous electrical activation in a threedimensional model of ventricular electromechanics. *J. Cardiovasc. Electrophysiol., 14*, S196–S202.

Valitutti, S., Müller, S., Cella, M., Padovan, E., & Lanzavecchia, A. (1995). Serial triggering of many T-cell receptors by a few peptide-MHC complexes. *Nature, 375*, 148-151.

Varela, F. J., & Coutinho, A. (1991). Second generation immune networks. *Imm Today, 12*(5), 159-166.

Velten, K., Reinicke, R., & Friedrich, K. (2000). Wear volume prediction with artificial neural networks. *Tribology International, 33*, 731-736.

Venkatraman, S., & Yen, G. G. (2005). A generic framework for constrained optimization using genetic algorithms. *IEEE Trans Evolutionary Computation, 9*(4), 424-435.

Vetter, F., & McCulloch, A. (1998). Three-dimensional analysis of regional cardiac function: a model of rabbit ventricular anatomy. *Prog. Biophys. Mol. Biol., 69*, 157–183.

von Neumann, J. (1963). The general and logical theory of automata. In J. von Neumann, & A. H. Taub (Ed.), *John von Neumann: Collected Works* (Vol. V, pp. 288-326). Oxford: Pergamon Press.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata.* (A. W. Burks, Ed.) Urbana: University of Illinois Press.

Walker, D. C., & Southgate, J. (2009). The virtual cell candidate co-ordinator for 'middle-out' modelling of biological systems. *Briefings in bioinformatics, 10*, 450-461.

Walker, D. C., Hill, G., Wood, S. M., Smallwood, R. H., & Southgate, J. (2004). Agent-based computational modeling of wounded epithelial cell monolayers. *IEEE Trans. Nanobioscience, 3*, 153–163.

Watanabe, H., Sugiura, S., Kafuku, H., & Hisada, T. (2004). Multi-physics simulation of left ventricular filling dynamics using fluid–structure interaction finite element method. *Biophys. J., 87*, 2074–2085.

Waters, C. M., & Bassler, B. L. (2005). QUORUM SENSING: Cell-to-Cell Communication in Bacteria. *Annual Review of Cell and Developmental Biology, 21*, 319-346.

Whalley, J., Tuite, M., & Johnson, C. (2002). A virtual lab for exploring the [psi]+ yeast prion. In F. Valafar (Ed.), *Proceedings of the 2002 International*

Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences. Las Vegas: CSREA Press.

White, T., & Pagurek, B. (1998). Towards multi-swarm problem solving in networks. *Proc of the 3rd Int Conf on multi-agent systems (ICMAS'98)*, (pp. 333-340).

Widrow, B., & Hoff, J. M. (1960). Adaptive switching circuits. *1960 IRE Western Electric Show Conv. Rec., pt. 4*, (pp. 96–104).

Wierzchoń, S., & Kużelewska, U. (2002). Stable Clusters Formation in an Artificial Immune System. *1st International Conference on Artificial Immune Systems (ICARIS).* Canterbury.

Wilke, C. O., & Adami, C. (2002). The biology of digital organisms. *Trends in Ecology and Evolution, 17*, 528-532.

Wilson, W. (1996). Lotka's game in predator–prey theory: linking populations to individuals. *Theor. Popul. Biol., 50*, 368-393.

Wilson, W., de Roos, D., & McCauley, S. (1993). Spatial instabilities within the diffusive Lotka-Volterra system: individual-based simulation results. *Theor. Popul. Biol., 43*, 91-127.

Wishart, D. S., Yang, R., Arndt, D., Tang, P., & Cruz, J. (2005). Dynamic cellular automata: an alternative approach to cellular simulation. *In Silico Biol., 5*, 139-161.

Wolfram, S. (1983). Statistical Mechanics of Cellular Automata. *Reviews of Modern Physics, 55*(3), 601-644.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents – theory and practice. *Knowl Eng Rev*(10), 115–52.

Xia, L., Huo, M., Wei, Q., Liu, F., & Crozier, S. (2005). Analysis of cardiac ventricular wall motion based on a three-dimensional electromechanical biventricular model. *Phys. Med. Biol., 50*, 1901–1917.

Yannakakis, G. N. (2005). *AI in computer games: generating interesting interactive opponents by the use of evolutionary computation.* PhD thesis. University of Edinburgh, College of Science and Engineering, School of Informatics.

Yoichi Nakayama, A. K., & Tomita, M. (2005). Dynamic simulation of red blood cell metabolism and its application to the analysis of a pathological condition. *Theor. Biol. Med. Model., 2:18*.

Zebulum, R. S., Pacheco, M. A., & Vellasco, M. M. (2001). *Evolutionary electronics: automatic design of electronic circuits and systems by genetic algorithms.* CRC Press.

Zhang, L., Athale, C., & Deisboeck, T. (2006). Development of a Three-dimensional Multiscale Agent-based Tumor Model: Simulating Gene-protein Interaction Profiles, Cell Phenotypes and Multicellular Patterns in Brain Cancer. *J Theoret Biol, 244*, 96-107.

Zhang, L., Zhou, C., Liu, X., Ma, Z., Ma, M., & Liang, Y. (2003). Solving multi objective optimization problems using particle swarm optimization. *Proceedings of IEEE congress on evolutionary computation 2003 (CEC 2003)*, (pp. 2400-2405). Canbella, Australia.

Zhang, Z., & Friedrich, K. (2003). Artificial neural networks applied to polymer composites: a review. *Composites Science and Technology, 63*, 2029–2044.

Zhang, Z., Klein, P., & Friedrich, K. (2002). Dynamic mechanical properties of PTFE based short carbon fiber reinforced composites: experiment and artificial neural network prediction. *Composites Science and Technology, 62*(7-8), 1001-1009.

Zheng, Y., Ma, L., Zhang, L., & Qian, J. (2003). On the convergence analysis and parameter selection in particle swarm optimization. *Proceedings of international conference on machine learning and cybernetics 2003*, (pp. 1802-1807).

Zorzenon dos Santos, R., & Coutinho, S. (2001). Dynamics of HIV infection: a cellular automata approach. *Phys. Rev. Lett., 87*, 168102.

Zygourakis, K., & Markenscoff, P. (1996). Computer-aided design of bioerodible devices with optimal release characteristics: a cellular automata approach. *Biomaterials, 17*, 125–135.

APPENDICES

Appendix A UML notation

UML or Unified Modelling Language is a general purpose modelling language for object oriented programming, created by the Object Management Group. The language has many types of diagrams to depict many aspects of a software product. The diagrams used here are called class diagrams (UML, 2010), and are used to represent the relations of classes.

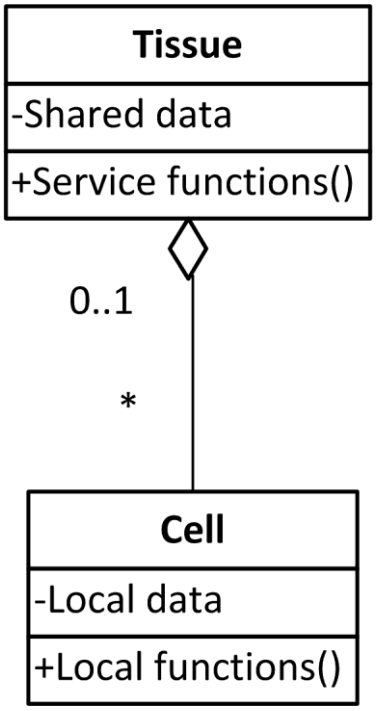
In the class diagram, each class is represented by a box with three parts, the topmost part shows the name of the class, the middle part lists the attributes of the class, sometimes denoted with a minus sign to differentiate it from the bottom part which consists of the operations of the class denoted with a plus sign.

There can be several types of associations between classes, the standard and probably most often used is a bi-directional association that is represented with a straight line. At both ends of the line, there is an indicator of multiplicity that indicates how many objects of the closer class can be associated with one object of the class at the other end of the association. This multiplicity can be either one number, like 0, 1, 5 or * (means any), or an interval of two values, like 1..* which means at least one.

There are many types of associations, but the ones used here are the aggregation types that represent the whole and its parts types of connections. These are denoted with a diamond shape. If the diamond is unfilled, it means that the relationship is basic aggregation that means that the child object can outlive the parent object. If the diamond is filled, the connection is a composition meaning that if a parent object is destroyed, the child object is destroyed with it.

The following example is Figure 1 showing two classes, the tissue consisting of shared data and operating with service functions and the cell consisting of local data and operating with local functions. The connection is a basic aggregation

showing that a maximum of one tissue can be associated with each cell, and there is no restriction of how many cells can be associated with a tissue.



Appendix B Supplementary materials

There are also some supplementary materials included on the DVD submitted together with the thesis. The following table details what can be found in the various folders on the disc.

Folder	Content
DeveloperDoc	Developer documentation for the framework and the cancer test
SourceFiles	Source code for the framework and a sample project (cancer test) created with the framework